Table des matières

1
La conception de logiciel

La baisse des coûts du matériel parallèlement à l'augmentation des performances a fait exploser le développement des applications et augmenter la demande en logiciel, à tel point que le système économique mondial est devenue irrémédiablement dépendant de la bonne marche de systèmes informatiques. L'utilisation de l'informatique se généralise à tous les domaines : transport aérien, banques, hôpitaux, cinéma, jeux, etc. Augmenter en proportion le nombre d'informaticiens pour répondre à cette demande ne sert à rien. Il est nécessaire d'appliquer une technologie en matière de conception et développement de (grands) systèmes logiciel, autrement dit en matière de génie logiciel.

Le génie logiciel n'est pas plus de « la programmation » que de « l'informatique ». Construire un système logiciel ne nécessite pas la seule utilisation d'un, voire de plusieurs langages de programmation. Les étapes préalables à la programmation proprement dite requièrent la mise en œuvre de méthodes de spécification et de conception, applications de théories issues d'autres domaines. Les deux sont indissociables.

Un logiciel n'est pas seulement l'ensemble des programmes informatiques associés à une application donnée, mais aussi la documentation nécessaire à la conception, l'installation, l'utilisation et la maintenance de ces programmes. Rien que le travail nécessaire pour rédiger la documentation est souvent aussi important que le travail de mise au point des programmes.

La demande de logiciel croît beaucoup plus vite que l'amélioration de la productivité. Nous avons besoin d'outils et de techniques puissants, ainsi que de formation des équipes chargées de la construction des systèmes logiciels. Il existe beaucoup d'outils. Nous avons choisi de présenter Java et UML.

1.1 Le cycle de vie du logiciel

L'ensemble des tâches nécessaires au développement et à la maintenance des logiciels est parfois appelé processus logiciel. Il ne peut malheureusement pas être décrit à l'aide d'un modèle simple et unique, car la nature, la complexité, l'organisation et l'enchaînement des tâches sont très variables avec les applications.

Si les modèles perfectionnés font toujours l'objet de recherches, un certain nombre de modèles généraux ont été mis au point (prototypage, transformation formelle, assemblage de composants réutilisables, etc.). L'avénement du génie logiciel en tant que discipline d'ingénierie à part entière amena l'élaboration d'un modèle de processus de développement de logiciel à partir d'expériences similaires dans d'autres domaines [**?**].

Le *modèle de la cascade*, le premier à avoir été mis au point, reste sans doute encore le plus utilisé, bien qu'il ne soit approprié qu'à certaines classes de systèmes logiciels. Il décrit le processus logiciel comme la descente en cascade d'une phase à une autre [**?**] :

– *Définition et analyse des besoins*. Les services du système, ses contraintes et ses objectifs sont établis en consultation avec les utilisateurs/clients. Ils sont

définis de manière à être compréhensibles à la fois par les utilisateurs et par l'équipe de programmation.
- *Conception du système et du logiciel*. Une architecture d'ensemble du système est défini à partir des besoins en logiciel et en matériel clairement identifiés. Les fonctions du système sont ensuite représentées de manière à être facilement transformables en un ou plusieurs programmes exécutables.
- *Implémentation et tests unitaires*. La conception est implémentée en un ensemble de programmes (unités de programmation). Chaque unité est testée séparément, pour vérifier que son implémentation correspond à sa spécification.
- *Intégration et tests du système*. Les unités sont intégrées en un système complet, qui, une fois testé, est livré au client.
- *Mise en œuvre et maintenance*. Le système est installé et mis en service. La maintenance comprend la corrections des bogues jusqu'alors non détectées, l'amélioration de l'implémentation, ainsi que l'enrichissement et l'adaptation du système au fur et à mesure que de nouveaux besoins apparaissent ou que les conditions d'exploitation changent.

Tout n'est pas si simple, car, dans la pratique, ces phases se chevauchent et interagissent. Par exemple, les problèmes correspondant aux besoins sont bien souvent identifiés au cours de la conception, ceux de la conception lors de l'implémentation, etc., nécessitant une série d'aller-retours entre les différentes phases

1.2 Analyse et conception

Il reste à trouver *comment* obtenir des modules avec les qualités requises. Un système logiciel, ou, pour simplifier, un programme, permet d'effectuer des traitements sur des données, comme le résume la célèbre formule [**?**] :

Algorithmes + Structures de données = Programmes

D'une manière générale, un concepteur a donc le choix entre les traitements et les données pour établir les bases de l'architecture de son programme.

1.2.1 Conception par traitements et conception par objets

La méthode de conception par traitements la plus classique est sans doute l'analyse fonctionnelle descendante, qui a déjà été évoquée dans le paragraphe **??** : elle procède par affinements successifs, en divisant récursivement la tâche à accomplir en sous-tâches moins complexes, jusqu'à obtenir des traitements suffisamment simples pour qu'ils puissent être directement implantés dans le langage choisi. La méthode revient à développer un arbre de tâches, où le niveau d'abstraction des tâches va décroissant, de la racine vers les feuilles

Ce genre de méthode a l'avantage de fournir au programmeur un bon guide pour amorcer la conception de son système logiciel. Elle est relativement facile à apprendre et à appliquer, car elle fait appel à un raisonnement logique que chacun sait

utiliser naturellement. Elle ne favorise cependant pas certaines des qualités importantes mises en exergue précédemment, telles que la réutilisabilité et l'extensibilité :

- Les modules trouvés ne sont naturellement pas généraux mais adaptés aux sous-problèmes pour lesquels ils ont été conc,us : dans la structure d'arbre obtenue, un fils pourvoit à un des besoins spécifiques de son père. La démarche descendante ne favorise donc pas la production d'éléments réutilisables, qui sont plutôt obtenus par composition de modules existants, c'est-à-dire par conception ascendante.
- Un programme est le plus souvent destiné à subir de nombreuses modifications tout au long de son exploitation. Par exemple, un logiciel conc,u à l'origine pour éditer des feuilles de paye mensuelles sera ensuite modifié pour produire également les feuilles de paye des ouvriers journaliers d'une nouvelle filiale, puis les bulletins annuels des revenus imposables. L'extensibilité du programme n'est guère favorisée par une architecture fondée sur les traitements à effectuer, qui sont moins stables que les structures de données à manipuler. Les relations temporelles et les interactions entre les traitements sont privilégiés dès le début de la conception, alors qu'il existe souvent de nombreuses solutions possibles, qui sont difficiles à choisir à ce moment, mais peuvent l'être ultérieurement, grâce à des expérimentations avec un prototype. Il est préférable de concentrer le travail de conception sur les composants du système, pour conserver un maximum de souplesse, en se dégageant des contraintes de séquencement et d'interfaçage des modules.
- La méthode ne tient pas compte des structures de données, dont les descriptions sont partagées entre les différents modules qui les utilisent. Comme dans les bibliothèques, le lien fédérateur entre les routines exploitant une même structure, c'est-à-dire la structure elle-même, est perdu.

Certes, une utilisation soigneuse de la méthode permet d'en pallier certains des défauts, mais la généralité n'est pas le propre de la méthode. Son principal mérite est d'être facile à comprendre et à appliquer, ce qui en fait un outil idéal pour l'enseignement. Elle reste cependant inadaptée à la conception de gros logiciels, notamment parce qu'elle suppose qu'au niveau d'abstraction le plus élevé, un logiciel peut être décrit de manière satisfaisante par une fonction unique. Or, c'est malheureusement bien souvent chose impossible, notamment avec un logiciel offrant plusieurs services, comme un système d'exploitation ou un tableur.

Une méthode de conception privilégiant les données conduit à définir l'architecture d'un logiciel à partir des données — des objets — qu'il traite, et non plus à partir de la fonction qu'il réalise. Le concepteur analyse les familles d'objets manipulés par le logiciel à construire et étaye la construction sur les améliorations successives résultant d'une meilleure compréhension de ces familles. De cette façon, la description et l'implantation des fonctions de plus haut niveau sont retardées le plus longtemps possible. La notion d'ordre ne prime pas : le concepteur recense les opérations applicables aux familles d'objets et précise leur déroulement, mais diffère autant que

possible la spécification de l'ordre d'application de ces opérations. Deux étapes principales sous-tendent cette démarche :

– Trouver les objets : ils correspondent aux objets de la réalité physique ou abstraite dans laquelle le logiciel doit opérer et qui se trouve décrite dans le cahier des charges. Donner une méthode systématique permettant d'établir de prime abord les bonnes familles d'objets est difficile. Il est certain que l'expérience du concepteur joue un rôle important en la matière.
– Décrire les objets : il ne s'agit pas de décrire *un* objet, mais une famille d'objets, c'est-à-dire une famille de structures de données qui ont des propriétés communes et dont l'objet est un représentant. La description d'une famille ne peut être axée sur l'implantation physique de ses objets. Ainsi, une pile ne doit pas être vue comme, par exemple, un tableau avec une capacité et un indice désignant le sommet de la pile. Une telle description manque par trop d'abstraction et de souplesse, puisque la représentation physique est susceptible d'être modifiée au cours de l'évolution du logiciel.

La solution à ce dernier problème est apportée par les types abstraits de données [**?**] [**?**], qui permettent de décrire des familles de structures de données en termes de services — d'opérations — offerts aux clients et de propriétés de ces services. La figure 1.1 montre comment une pile pourrait être décrite à l'aide d'un formalisme de type abstrait :

– La rubrique **Type** introduit le type, qui est ici générique. `T` est un paramètre formel qui, une fois instancié, permet d'obtenir un type de pile effectif, par exemple une pile d'entiers : `PILE[ENTIER]`. Nul besoin donc d'écrire des descriptions séparées et quasiment identiques pour les piles d'entiers, les piles de caractères, les piles de salariés, etc.
– La rubrique **Fonctions** énumère les services disponibles sur les objets représentant les occurrences du type `PILE`. Chaque service est exprimé par une fonction mathématique dont est donnée la signature. La fonction de construction fournit une nouvelle occurrence du type : elle correspond à la *création* d'un objet. Les fonctions d'*accès* opèrent sur une occurrence du type et fournissent une valeur d'un type autre : elles correspondent à la consultation des données — de l'état — d'un objet. Les fonctions de *transformation* fournissent une nouvelle occurrence du type à partir de l'occurrence sur laquelle elles sont appliquées : elles correspondent à la modification de l'état d'un objet.
– Certaines fonctions, dites partielles et indiquées par une flèche barrée, ne sont pas définies pour toute occurrence du type. La rubrique **Préconditions** en précise les conditions d'application : ainsi, il n'est possible ni de retirer un élément ni de consulter le sommet d'une pile vide.
– Les divers services doivent respecter les propriétés d'une pile, qui ne peuvent pas être décrites par la seule donnée de fonctions et qui sont donc spécifiées dans la rubrique **Axiomes**. En particulier, les deux dernières lignes énoncent

Type
PILE[T]

Fonctions
fonctions de construction
créer: $\rightarrow$ PILE[T]
fonctions d'accès
vide: PILE[T] $\rightarrow$ BOOLÉEN – *La pile est-elle vide ?*
sommet: PILE[T] $\not\rightarrow$ T – *Consulter le sommet de la pile.*
fonctions de transformation
empiler: PILE[T] $\times$ T $\rightarrow$ PILE[T]
dépiler: PILE[T] $\not\rightarrow$ PILE[T]

Préconditions
$\forall$ p:PILE[T]
dépiler(p) **est défini ssi non** vide(p)
sommet(p) **est défini ssi non** vide(p)

Axiomes
$\forall$ t:T, p:PILE[T]
vide(créer()) = VRAI
vide(empiler(p,t)) = FAUX } *Propriétés de la fonction* vide
sommet(empiler(p,t)) = t
dépiler(empiler(p,t)) = p } *Principe dernier–entré–premier–sorti*

FIG. 1.1 – *Description simplifiée d'une pile à l'aide d'un un type abstrait de données (d'après [**?**]).*

le principe de gestion d'une pile, dernier–entré–premier–sorti : un nouvel élément se place au sommet de la pile et empiler un nouvel élément puis le dépiler donne la pile initiale. Faute de ces contraintes, la description proposée engloberait aussi des structures comme les listes ou les files.

Cette approche conduit à une méthode de conception de logiciel organisant chaque module autour d'une famille de structures de données, dont la description est celle d'un type abstrait de données. C'est un point de vue opérationnel, qui est en conformité avec les principes d'abstraction de données et de masquage d'information. Un module ne s'occupe que de ses propres affaires et ses clients n'en ont qu'une vision externe : ils n'accèdent aux données du module qu'à travers un ensemble de services dûment répertoriés, et non pas sur la base d'une implantation fixée à un moment particulier de l'évolution du logiciel. Ce point de vue est le seul compatible avec le développement intensif de logiciel puisqu'il permet de préserver l'intégrité de chaque composant du logiciel dans un contexte de changements fréquents. L'architecture du logiciel est fondée sur la structure des données, qui est elle-même exprimée en termes de fonctions abstraites. La méthode de conception est guidée par les données, mais elle installe les fonctions à leur juste place.

FIG. 1.2 – *L'objet* `p1`*, instance de la classe* `POINT`*, symbolisé par une calculette.*

Toutefois, l'objectif de la méthode est bien la conception et l'implantation de logiciel, et non pas la simple spécification. Un module est donc l'*implantation* d'un type abstrait de données, le type lui-même n'étant qu'une spécification. Une méthode visant la construction de logiciel à partir de tels modules doit aller de pair avec un langage de programmation permettant d'implanter les modules.

Pour résumer, cette méthode de conception permet un développement fondé sur une spécification précise de classes en termes de besoins (services offerts aux clients) et de contraintes (propriétés de ces services). Un objet est défini, donc connu, par son comportement (les services offerts), indépendamment de sa représentation physique. Les objectifs exprimés précédemment sont atteints :

– Abstraction de données : les raisonnements sous-jacents à l'analyse du problème à traiter sont effectués en termes de concepts abstraits, qui sont ensuite modélisés *directement* par des classes. La généricité accroît encore la puissance d'abstraction de données.
– Modularité et compatibilité : le logiciel est découpé en modules, qui peuvent être compilés séparément et groupés en bibliothèques.
– Réutilisabilité : l'héritage permet de définir incrémentalement de nouvelles classes à partir des classes existantes.
– Extensibilité : une classe étant définie en termes de services, une modification de la structure de ses objets n'a que peu ou pas d'incidence sur la manière dont les clients de la classe utilisent ces objets.
– Lisibilité : l'interface de la classe en donne un mode d'emploi clair et précis. Le code source est d'autant plus lisible que les données des objets sont manipulées grâce aux opérations déclarées dans l'interface, l'implantation physique restant cachée.

1.2.2 Les notions de classe et d'objet

Il est temps d'aborder les notions de classe et d'objet sur des bases simplifiées, plus pragmatiques et plus générales, et d'introduire quelques éléments du vocabulaire propre au domaine.

De façon imagée, nous pouvons considérer un objet comme une calculette dont les touches symbolisent les opérations qui peuvent lui être appliquées. La figure 1.2 schématise sous cette forme l'objet `p1`, représentant un point du plan cartésien. L'action d'appuyer sur une touche provoque l'exécution de l'opération correspondante. Si un résultat doit être délivré, il apparaît ensuite dans la fenêtre d'affichage de la calculette. Par exemple, les touches `x` et `y` permettent de consulter l'abscisse et l'ordonnée courantes du point, qui valent respectivement `100.` et `25.` Après l'exécution de l'opération `en_haut` (3), l'abscisse reste inchangée (4) mais l'ordonnée est augmentée d'une unité (5).

interface de la classe `POINT`

méthodes d'accàes

`x : → RÉEL`	– *Consulter l'abscisse du point.*
`y : → RÉEL`	– *Consulter l'ordonnée du point.*

méthodes de transformation

`en_haut`	– *Ajouter 1 à l'ordonnée du point.*
`à_droite`	– *Ajouter 1 à l'abscisse du point.*
`en_bas`	– *Retirer 1 à l'ordonnée du point.*
`à_gauche`	– *Retirer 1 à l'abscisse du point.*
`déplacer(dx : RÉEL, dy : RÉEL)`	– *Déplacer le point de 'dx' unités en*
	– *abscisse et 'dy' unités en ordonnée.*

FIG. 1.3 – *L'interface de la classe* `POINT`.

Le principe du masquage d'information est bien respecté. Pour paraphraser la formule donnée en prologue au paragraphe **??**, nous pouvons dire :

Opérations + Structures de données privées = Objets

Le client n'a nul besoin de connaître la mécanique interne de la calculette pour modifier et consulter les données. Il lui suffit de presser les touches adéquates. Il connaît précisément l'ensemble complet des services offerts par l'objet au vu du clavier, qui est le reflet de l'interface spécifiée par la classe de l'objet. Celle de la classe `POINT` est montrée à la figure 1.3. Dans le vocabulaire spécialisé, une opération est souvent désignée par le terme générique de *méthode*. Sept méthodes, donc, sont disponibles. Comme nous l'avons montré à la figure **??**, les deux premières, `x` et `y`, permettent de connaître les valeurs de l'abscisse et de l'ordonnée d'un point. Elles correspondent aux fonctions d'accès de la définition d'un type abstrait (cf. Fig. 1.1). Elles peuvent d'ailleurs être assimilées à des fonctions, au sens du langage Pascal, dans la mesure où elles retournent une valeur, ici un nombre réel.

Les cinq autres méthodes servent à déplacer le point dans le plan. La dernière, `déplacer`, admet deux paramètres réels, la valeur du déplacement en abscisse et la valeur du déplacement en ordonnée, respectivement. Ces méthodes correspondent aux fonctions de transformation du type abstrait, qui retournent une occurrence du type : ainsi, le résultat des fonctions `empiler` et `dépiler` est une nouvelle pile dont le sommet est modifié (cf. Fig. 1.1). Toutefois, le type abstrait est un formalisme mathématique conc,u pour décrire rigoureusement les opérations et leurs propriétés au travers d'une spécification fonctionnelle. Son principe ne peut être totalement respecté lors de la phase d'*implantation*, qui permet de passer du type abstrait à la classe. Cela impliquerait, entre autres, que chaque méthode de transformation retourne une copie modifiée de l'objet sur lequel elle opère, entraînant un gâchis de ressources inconcevable en pratique. Du point de vue mathématique, le problème ne se pose pas puisque la notion d'espace mémoire n'existe pas. Du point de vue pratique, il est réglé par l'utilisation d'effets de bord. Les cinq méthodes sont assimilées à des procédures, qui modifient les valeurs de l'abscisse et de l'ordonnée du point en intervenant

```
Classe
  POINT

Variables d'instance
  abscisse : RÉEL;
  ordonnée : RÉEL;

Méthodes
  fonction x : RÉEL
    retourner(abscisse);
  fin fonction

  fonction y : RÉEL
    retourner(ordonnée);
  fin fonction

  procédure en_haut
    ordonnée := ordonnée + 1;
  fin procédure

  procédure à_droite
    abscisse := abscisse + 1;
  fin procédure

  procédure en_bas
    ordonnée := ordonnée - 1;
  fin procédure

  procédure à_gauche
    abscisse := abscisse - 1;
  fin procédure

  procédure déplacer(dx : RÉEL,dy : RÉEL)
    abscisse := abscisse + dx;
    ordonnée := ordonnée + dy;
  fin procédure
```

FIG. 1.4 – *Définition de la classe* `POINT`*.*

directement dans la structure de données représentant le point, sans la dupliquer : ces méthodes ne retournent donc pas de résultat.

La description proprement dite de la classe est le fait du concepteur, qui choisit la structure de données pour représenter les objets et qui écrit le corps des fonctions — au sens informatique du terme — et des procédures correspondant aux méthodes. La structure est définie par un ensemble de variables, comme le montre la figure 1.4, qui donne le texte de la classe `POINT` dans une syntaxe imaginaire voulue la plus lisible possible. En l'occurrence, le concepteur a décidé de représenter un point par deux valeurs réelles, figurant son abscisse et son ordonnée.

Le modèle de point ainsi défini sert à construire les objets qui sont les occurrences, ou encore les instances, de la classe `POINT`. Un objet est créé par instanciation de sa classe, grâce à une opération spéciale dont l'usage et la nature diffèrent selon les langages. La figure 1.5, met en évidence le modèle des objets dans la dé-

FIG. 1.5 – *L'instance* `p1` *est créée sur le modèle défini par sa classe,* `POINT`.

```
  abs, ord: RÉEL;
– Variable pour désigner un objet de la classe POINT
  p1: POINT;
  ...
– Créer un objet de la classe POINT d'abscisse 100 et d'ordonnée 25.
  p1 := créer(100.,25.);
– Déplacer le point de 10 unités en abscisse et de 20 unités en ordonnée.
  p1.déplacer(10.,20.);
– Stocker les nouvelles coordonnées dans deux variables.
  abs := p1.x;
  ord := p1.y;
  ...
```

FIG. 1.6 – *Un exemple de texte source avec des envois de messages.*

finition de la classe, où se retrouvent les trois parties de la figure 1.4 : le nom de la classe, les variables et les méthodes. L'opération d'instanciation consiste en quelque sorte à mouler un objet d'après la structure définie par les variables [**?**]. Pour cette raison même, celles-ci sont appelées variables d'instance. Les emplacements de la structure moulée sont ensuite remplies avec les valeurs adéquates, ici les coordonnées du point. La structure symbolisant l'objet est pourvue d'une étiquette rappelant le nom de la classe dont il est instance.

Une fois qu'un objet est créé, ses clients ne peuvent le manipuler qu'à l'aide des méthodes de l'interface. C'est l'objet lui-même qui reste responsable de la manière dont les actions correspondantes sont effectuées. L'invocation d'une méthode est donc plutôt une requête, un *message*, envoyé à l'objet par un client pour demander l'exécution d'une certaine opération : les objets et leurs clients communiquent par envois de messages. Un message stipule le nom de la méthode correspondant à l'opération à effectuer, les arguments effectifs éventuels de cette méthode et, bien sûr, le destinataire du message, que nous conviendrons d'appeler *objet receveur* , d'après la terminologie employé en Smalltalk. La syntaxe d'un envoi de message prend des formes très variées selon le langage de programmation utilisé. Eiffel, comme C++ et d'autres langages apparentés, utilise une notation pointée où la cible, qui est une expression désignant l'objet receveur, est suivie d'un point introduisant le nom de l'opération à effectuer et les éventuels arguments effectifs. Par exemple, le texte de la figure 1.6 montre comment créer une instance de la classe `POINT` en utilisant une primitive spécifique du même langage imaginaire que précédemment, la fonction `créer` (ligne 1). Suivent trois envois de messages. Le premier (ligne 2) est sans retour, car il invoque une méthode de transformation, équivalente à une procédure, pour déplacer le point dans le plan. Il a pour effet de modifier les valeurs des variables d'instance de l'objet `p1`. Les deux suivants (lignes 3 et 4) sont des messages avec retour, puisqu'ils invoquent des méthodes d'accès, équivalentes à des fonctions.

Les résultats transmis, représentant les nouvelles coordonnées du point, sont stockés dans les deux variables locales `abs` et `ord`.

Si, pour des raisons pédagogiques, nous avons considéré d'abord l'objet, avant la classe qui en donne la description, il est entendu que la classe doit exister avant l'objet. L'écriture d'un programme commence bien par la définition des classes modélisant l'univers de l'application traitée. Les objets nécessaires sont ensuite créés par instanciation des classes et les actions à effectuer sont accomplies en envoyant les messages adéquats aux objets.

1.3 Exécuter un programme

Pour être opérationnel plus rapidement et passer immédiatement à du concret, plaçons-nous d'emblée en situation de travaux pratiques, comme si nous étions « devant la machine ». L'objectif est d'expliquer comment fonctionnent la compilation et l'exécution d'un programme Java très rudimentaire, de manière à ce que le lecteur puisse essayer une partie des exemples au fur et à mesure de son avancement.

1.3.1 La machine virtuelle Java

Le composant central de l'environnement Java est sa machine virtuelle. Pour bien comprendre en quoi elle consiste, revenons à des machines qui sont sûrement plus attractives, vu leur succès actuel : les consoles de jeu, dont les *Gameboy* et autres *PlayStation* sont des exemples bien connus.

Des jeux prévus pour une console peuvent quelquefois se pratiquer sur un ordinateur personnel grâce à un programme appelé émulateur, qui simule le fonctionnement de la console et permet donc d'utiliser n'importe quel programme de jeu s'exécutant sur la dite console. Tout se passe comme si l'on disposait physiquement de la console, bien que celle-ci n'existe, en fait, qu'à l'état virtuel dans la machine hôte. La même technique peut être utilisée pour, par exemple, faire fonctionner des programes prévus pour l'environnement Windows sur un ordinateur Macintosh.

En ce qui concerne la machine virtuelle Java, il y a juste une petite différence : elle émule une machine qui n'a jamais été construite et qui n'existe que sous la forme d'une spécification sur papier, la machine en question méritant pleinement son qualificatif de « virtuelle », cette fois. La spécification, qui décrit le langage d'assemblage et l'architecture générale de cette machine virtuelle [**?**], a été mise à la disposition du public. N'importe quelle personne suffisamment compétente peut ainsi écrire un programme simulant la machine virtuelle sur une machine réelle, comme un Macintosh, mais aussi sur des supports plus surprenants, comme un système embarqué dans une voiture, un PalmPilot, un téléphone portable, voire, pourquoi pas, une machine à laver, une chaîne hi-fi, etc.

Cette approche n'est pas récente, puisqu'elle a déjà été utilisée pour le système Smalltalk dans les années 1970 ou l'interprète Le_Lisp dans les années 1980 Le

pcode du langage Pascal relève aussi du même genre d'idée. Il est tout à fait possible de « réaliser » une machine virtuelle, et l'expérience a d'ailleurs été tentée plusieurs fois pour Java, en construisant une véritable machine, la *Java Station*, sans réel succès commercial, ou encore en implantant la machine virtuelle sur un processeur spécialisé, avec PicoJava ou JSCP pour des systèmes embarqués.

Plus concrètement, une fois le code source d'une application écrit en langage Java :

- Un compilateur traduit les instructions Java dans le langage, plus primitif, de la machine virtuelle, qui est indépendant de toute architecture matérielle.
- La machine virtuelle Java de la machine cible charge le programme sous sa forme compilée et l'exécute.

Le même code compilé, donc la même application, peut ensuite être exécuté tel quel, sans aucune modification, sur toute autre architecture implantant la machine virtuelle. Avec des langages comme C ou C++, il faudrait procéder à une nouvelle compilation, voire à des modifications du code source pour l'adapter aux caractéristiques de la nouvelle architecture.

Cet avantage est des plus appréciables et constitue sans doute le principal intérêt de Java, quand on sait que les portages sont le plus souvent d'un coût très élevé en raison des problèmes très techniques qu'ils soulèvent. Imaginez le travail que représente l'écriture d'un navigateur qui doit fonctionner sur MacOS, les différentes versions de Windows et les multiples versions d'UNIX, pour ne parler que des systèmes les plus répandus.

1.3.2 Compilation et exécution d'un programme

Passons immédiatement à la pratique, en compilant puis exécutant le célébrissime programme *hello world* qui, comme l'indique son nom, ne fait qu'imprimer cette simple phrase : *hello world*. Nous sommes supposés travailler, et ceci restera vrai pour toute la suite, sur une station Sun, sous le système d'exploitation UNIX. Le caractère d'invite de l'interpréteur de commandes, le *shell* UNIX, est un '`%`'.

1. Le code source est saisi sous la forme d'un ensemble de fichiers contenant les définitions des classes nécessaires à l'implantation du programme.
   UNE CLASSE PAR FICHIER ?
   Le nom d'un tel fichier doit comporter l'extension `.java`. Voici donc le contenu du fichier `Hello.java` correspondant à notre programme :
2. Chaque fichier est ensuite compilé pour produire le code exécutable, conforme à la spécification du langage d'assemblage défini pour la machine virtuelle [**?**]. Le compilateur livré avec le kit de développement Java d'une station de travail Sun s'appelle `javac` :

```
...
% javac Hello.java
%
```

3. La compilation d'un unique fichier source produit autant de fichiers d'extension `.class` qu'il y a de classes définies dans le fichier source. Chacun de ces fichiers résultats porte *le nom de la classe* correspondante (et non pas celui du fichier source). Le code qu'il contient est sous forme numérique, mais il peut être examiné sous une forme plus lisible, pour qui connaît le langage de la machine virtuelle, grâce à un désassembleur, appelé `javap` :

   ```
   ...
   % javap -c Hello
   Compiled from Hello.java
   class Hello extends java.lang.Object {
       Hello ();
       public static void main(java.lang.String []);
   }

   Method Hello()
      0 aload_0
      1 invokespecial #6 <Method java.lang.Object()>
      4 return

   Method void main(java.lang.String [])
      0 getstatic #7 <Field java.io.PrintStream out>
      3 ldc #1 <String "hello␣world">
      5 invokevirtual #8 <Method void println(java.lang.String)>
      8 return
   %
   ```

   Sans chercher à comprendre le contenu et le rôle des instructions, on observe bien que le corps des méthodes a été traduit dans un langage plus primitif que Java : `ldc`, `invokevirtual`, `return`, etc. sont des instructions de la machine virtuelle.

4. L'une des classes du programme doit obligatoirement contenir une méthode appelée `main`, qui constitue le point d'entrée du programme et doit avoir le profil suivant :

   ```
   class unExemple {
     public static void main(String args []) {
       // ...
     }
   }
   ```

   Le programme s'exécute en fournissant le nom de la classe en question à la machine virtuelle, invoquée grâce à la commande `java` :

   ```
   ...
   % java Hello
   ```

   Attention : ici, `Hello` est bien le nom d'une classe et non celui d'un fichier.

5. La variable d'environement `CLASSPATH` contient la liste des répertoires où se trouvent les fichiers de code exécutable (d'extension `.class`). La machine virtuelle parcourt ces répertoires, dans l'ordre, jusqu'à trouver un fichier conte-

nant le code compilé de la classe qui lui est donnée en paramètre, dans notre cas, donc, un fichier appelé `Hello.class`.
Si la variable `CLASSPATH` n'est pas positionnée, la machine Java cherche simplement dans le répertoire courant (toujours dans le cas d'une station de travail Sun).

6. L'exécution démarre alors à la première instruction de la méthode `main`. Pour ce premier exemple, le résultat serait le suivant :

```
...
% java Hello
hello world
%
```

CA NE VA PAS : CA NE CORRESPOND PAS A CE QUI EST DIT !
La figure 1.1

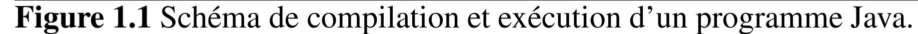
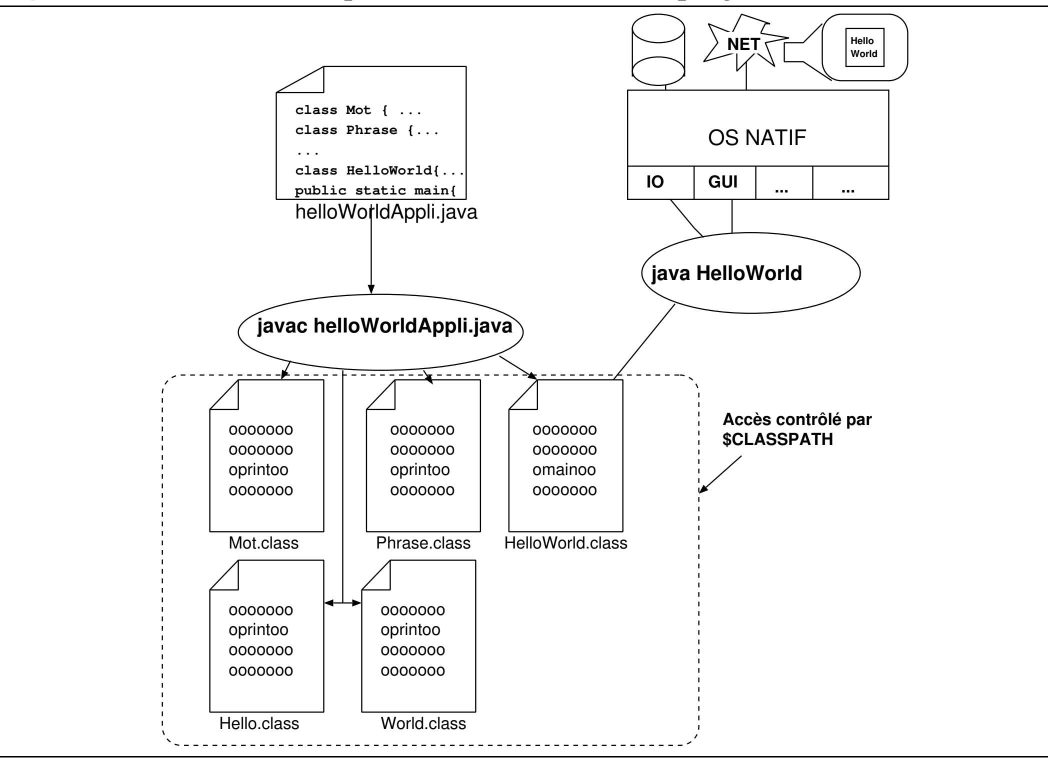


Figure 1.1 Schéma de compilation et exécution d'un programme Java.



résume graphiquement le processus de compilation et d'exécution du programme Java.

1.3.3 La bibliothèque Java

Il est certain que cette vision du cycle de compilation et exécution est assez simpliste. En particulier, le programme que nous venons d'exécuter a écrit *hello world*

sur la sortie standard de l'interprète de commandes de la station de travail. Or, pour réaliser une entrée/sortie, il faut disposer des objets permettant de représenter les mécanismes d'entrée/sortie, donc des classes décrivant ces objets.

Ces classes, et quelques autres, sont regroupées dans la bibliothèque Java, et elles ont été chargées par la machine virtuelle en même temps que la classe de l'application, `Hello`. Le lancement du programme en mode « verbeux » (option `-verbose`) permet de s'en rendre compte aisément, car il provoque l'affichage des noms de toutes les classes chargées pour l'exécution (Fig. 1.2).

Figure 1.2 Les classes chargées lors de l'exécution du programme *hello world*.

```
...
% java -verbose Hello
[Opened /local/langages/Java/ solaris /jdk1 .2.2/ jre / lib / rt . jar  in  40 ms]
[Opened /local/langages/Java/ solaris /jdk1 .2.2/ jre / lib /i18n. jar  in  4 ms]
[Loaded java . lang .NoClassDefFoundError from /local/langages/Java/ solaris /jdk1 .2.2/ jre / lib / rt . jar ]
[Loaded java . lang .Class  from / local /langages/Java/ solaris /jdk1 .2.2/ jre / lib / rt . jar ]
[Loaded java . lang .Object  from / local /langages/Java/ solaris /jdk1 .2.2/ jre / lib / rt . jar ]
[Loaded java . lang .Throwable from / local /langages/Java/ solaris /jdk1 .2.2/ jre / lib / rt . jar ]
[Loaded java . io .  Serializable   from / local /langages/Java/ solaris /jdk1 .2.2/ jre / lib / rt . jar ]
[Loaded java . lang . String  from / local /langages/Java/ solaris /jdk1 .2.2/ jre / lib / rt . jar ]
...
...
[Loaded Ljava. security . cert .  Certificate  ;]
[Loaded Hello]
hello  world
%
```

En fait, l'exécution du programme *hello world*, qui est le plus rudimentaire qui soit, nécessite pas moins de cent soixante dix classes de la bibliothèque Java ! Mais il n'y a pas eu besoin de les compiler, car elles l'étaient déjà, comme toutes celles de la bibliothèque. Elles ont, en quelque sorte, été utilisées « en l'état ». Autrement dit, il n'est nul besoin du code source, seul le code compilé est nécessaire, comme dans toute approche à base de compilation séparée. C'est également le cas avec les bibliothèques C ou C++, par exemple.

Ce fait est très important dans le cadre d'une approche industrielle de la distribution de logiciel. En effet, supposons que vous écriviez une application Java destinée à être commercialisée, un moteur d'animation 3D, par exemple. Vous pouvez alors vendre uniquement la bibliothèque compilée implantant le moteur, sans avoir à révéler vos secrets de fabrication en diffusant le code source correspondant.

Remarquez également la machine Java a trouvé où sont localisées les classes de la bibliothèque : le répertoire de la bibliothèque Java est, par défaut, ajoutée à la liste contenue dans la variable `CLASSPATH`. Par exemple, la ligne 9 de la figure 1.2 montre que la classe String a été chargée à partir d'un fichier `rt.jar`.

Comme les classes de la bibliothèque sont fort nombreuses, elles sont compressées dans des fichiers d'archive d'extension `.jar`, selon le même principe que celui

de l'outil `zip`, disponible, entre autres, sous Windows. Il est possible de connaître le contenu d'un fichier d'archive grâce à l'outil `jar` (Fig. 1.3),

Figure 1.3 Le contenu du fichier d'archive `rt.jar`.

```
...
% jar −tvf /local/langages/Java/ solaris /
jdk1 .2.2/ jre / lib / rt . jar
     0 Tue Jun 29 05:00:36 CEST 1999 META−INF/
  2177 Tue Jun 29 05:00:36 CEST 1999 META−INF/MANIFEST.MF
     0 Tue Jun 29 03:48:34 CEST 1999 com/
     0 Tue Jun 29 04:59:50 CEST 1999 com/sun/
     0 Tue Jun 29 04:03:06 CEST 1999 com/sun/java/
     0 Tue Jun 29 04:03:06 CEST 1999 com/sun/java/swing/
     0 Tue Jun 29 04:03:28 CEST 1999 com/sun/java/swing/plaf/
     0 Tue Jun 29 04:03:48 CEST 1999 com/sun/java/swing/plaf/motif/
  1348 Tue Jun 29 04:03:06 CEST 1999 com/sun/java/swing/plaf/motif/ MotifButtonListener . class
  2814 Tue Jun 29 04:03:06 CEST 1999 com/sun/java/swing/plaf/motif/MotifButtonUI. class
  1684 Tue Jun 29 04:03:08 CEST 1999 com/sun/java/swing/plaf/motif/MotifBorders. class
  ...
  ...
```

dont les différentes options sont décrites dans la documentation accompagnant le kit de développement Java.

Les archives permettent de distribuer plus facilement et plus rapidement les bibliothèques et accélèrent la recherche des fichiers de classes. Leurs chemins d'accès peuvent être indiqués directement dans la variable `CLASSPATH`. D'une manière générale, c'est le programmeur qui fixe lui-même la valeur de la variable `CLASSPATH`, en fonction des répertoires où sont stockés les fichiers de code compilé de ses applications. Dans notre exemple, la variable pourrait prendre la valeur donnée à la figure 1.4.

Figure 1.4 Fixer la valeur de la variable d'environnement `CLASSPATH`.

```
...
% setenv CLASSPATH .:Java/Livre/Classes−et−objets/compile/exemple1
% cd
% java Hello
hello world
%
```

La machine recherche donc d'abord le fichier `Hello.class` dans le répertoire courant (`.`), dont il ne fait pas partie, puis dans le répertoire `Java/Livre/Classes-et-objets/comp` du répertoire courant, où elle le trouve. Il est alors chargé et exécuté.

Pour finir, notez que la variable `CLASSPATH` n'est pas seulement utilisée au moment du lancement de l'exécution d'un programme, mais également au cours même

de l'exécution. En effet, la machine virtuelle Java est capable de charger dynamiquement les classes dont elle a besoin au fur et à mesure qu'elle exécute un programme, et de continuer l'exécution par le code nouvellement chargé. Nous y reviendrons.

2
Cahier des charges : Tortues Java

2.1 Les tortues java

Le jeu se déroule sur un ensemble de cartes prédéfinies en forme de grilles rectangulaires, peuplées de monstres hostiles et belliqueux. Le héros doit amasser un maximum de richesses, et accessoirement découvrir l'Amulette de Java dissimulée dans la dernière carte, avant de quitter le jeu.

2.1.1 Les éléments du jeu

Les cartes

Il existe un nombre $nbCartes$, invariable, de *cartes* rectangulaires de dimensions identiques, découpées en $long \times larg$ *cases*. Une case contient un élément de *paysage*, herbe, arbre, ou eau, ces deux derniers constituant des obstacles infranchissables par les *acteurs* du jeu.

Une case franchissable (contenant de l'herbe) peut dissimuler une *porte*, permettant au héros, et uniquement au héros, de passer à la carte suivante. Une porte n'est visible qu'au joueur se trouvant sur la case qui la contient. Elle est à sens unique : elle ne permet pas de retourner à la carte précédente.

Une case franchissable peut également contenir un ou plusieurs objets : armes, potions magiques, or, sac. Un et un seul acteur peut occuper une case à un instant donné.

Les acteurs

Le *héros* est équipé d'une *armure*, d'un *bouclier*, d'une *arme* de poing et porte un sac de capacité illimité. Il est caractérisé par un *coefficient de force* $force$, des *points d'expérience* exp, qui croissent au fur et à mesure du jeu, et des *points de vie* hp.

De la même façon, un *monstre* est caractérisé par un *coefficient de force* $force$, des *points d'expérience* exp, et des *points de vie* hp. Il peut également porter une armure, une arme de poing et un sac.

Les différents monstres sont :

- L'Handray,
- Le Mhâziny...
- Le Mauhly...

Les acteurs se déplacent d'une case à la fois, sur une des huit cases adjacentes à celle où ils se trouvent à un instant donné. Si la case d'arrivée contient des objets, des armes, ou un sac, l'acteur peut les ramasser pour les mettre dans son sac. Si elle contient un autre acteur, le déplacement revient à porter un coup à ce dernier, en utilisant l'arme porté par l'acteur qui se déplace.

Franchir une porte permet au héros d'augmenter ses points d'expérience de 15 pour cent et de remonter sa vie au maximum.

Les armes

Une armure et un bouclier sont caractérisés par un *coefficient de protection* $prot$, une arme par un *coefficient de destruction* $destr$. Ces coefficients sont constants : ils ne peuvent pas varier au cours du jeu.

Il existe deux types d'armures, en airain et en acier, de coefficients respectifs 2 et 4, trois types de boucliers, en bois, en airain et en acier, de coefficients respectifs 1, 2 et 3, et trois types d'armes, la massue, le glaive et la hache, de coefficients respectifs 2, 4 et 6.

Les objets

Un sac a une *capacité* illimitée. Il permet de porter les différents objets et armes éventuellement ramassés au cours du jeu. Un sac peut contenir un autre sac ramassé au cours du jeu.

L'or est caractérisée par une valeur en *écus*.

Il existe 3 type potions magiques:

Vie Gain ou pertes de points de vie

Expérience Gain ou pertes de points d'expérience

Paralysie Paralysie momentanée du joueur sur un nombre de tour.

Les combats

Lorsqu'un acteur A porte un coup à un autre acteur B, les points de vie de B diminuent d'une quantité calculée en fonction des caractéristiques des deux acteurs, de la façon suivante :

$B.hp = B.hp - (A.degat - B.prot)$
$A.degat = A.force + A.arme.destr$
$B.prot = B.armure.prot + B.bouclier.prot$

Lorsque la quantité de points de vie d'un acteur devient négative ou nulle, l'acteur meurt et disparaît du jeu. Son sac ainsi que ses différentes armes (armure, bouclier, arme) tombent sur le sol, où ils pourront être ramassés par les autres acteurs.

Lorsqu'un acteur A gagne un combat contre un acteur B, ses points d'expériences augmentent de la moitié du nombre de points d'expérience de B.

Tous les 100 points d'expériences, un acteur augmente ses caractéristiques de vie et de force de 10 pour cent.

2.1.2 Le jeu

La construction des cartes

Il existe un nombre fixé de cartes prédéfinies comportant:

- Des cases constituant des obstacles infranchissables, les autres cases contenant de l'herbe.

- une porte et une seule,
- le héros, sur une position prédéterminée
- entre 2 et 5 monstres de chaque catégorie
- de 3 à 5 tas d'or, de valeur comprise entre 10 et 100 écus,
- de 0 à 2 armes de chaque catégorie (armure, bouclier, arme de poing)
- des potions magiques
- l'Amulette de Java, si la carte est la dernière.

Il peut y avoir plusieurs armes et objets sur une même case, mais un seul acteur.

Le déroulement du jeu

Le jeu se déroule en cycles, comprenant une action du héros (qui joue donc le premier), puis une action de chacun des monstres actifs de la carte courante, dans un ordre non défini.

Le héros a le choix entre quatre types d'action :

- se déplacer sur une des huits cases adjacentes,
- ramasser un des éventuels objets de sa case,
- franchir une porte, si la case sur laquelle il se trouve est une porte.
- Se reposer un tour pour regagner des points de vie
- boire une potion
- porter ou changer d'arme de points, d'armure ou de bouclier contenus dans le sac
- quitter le jeu.

Un monstre peut :

1. se déplacer sur une des huits cases adjacentes, au hazard sauf si le héro se trouve sur une des 8 cases adjacentes. Dans ce cas il frappe le heros en priorité.
2. Se déplacer sur une case contenant un autre monstre. Dans ce cas il y a combat entre les deux monstres.
3. ramasser un des éventuels objets de sa case.
4. porter une arme de poing, une armure ou un bouclier, celui qui a le plus fort coefficient parmi ceux dont il dispose.
5. boire une potion

Le jeu s'arrête lorsque le héros meurt ou lorsqu'il quitte volontairement le jeu. Un score est calculé en fonction de ses possessions à cet instant, de la façon suivante :

$$hp + exp + or + valeur\, en\, ecu\, de\, ses\, possessions$$

Le héros est ensuite inséré dans le classement répertoriant les 10 meilleurs scores réalisés par les utilisateurs de la version courante du jeu.

La partie graphique

- Vous avez libre choix pour réaliser le controle du héros humain (le clavier me semble un bon choix). Un écran d'aide doit pouvoir renseigner l'utilisateur sur les moyens de faire bouger le héros humain.

– Si l'utilisateur demande à terminer et sans que le héros humain soit mort, alors il faut confirmer la fin du jeu.
– Description du contenu du sac lorsque le héro porte une arme

2.2 Analyse

Nous avons réalisé avec les étudiants de l'Ecole Supérieure d'Informatique et d'Application de Lorraine ESIAL l'application"Tortue Java" conformément au cahier des charges.

Au cours de l'analyse, nous avons conçu les différents écrans dont nous aurons besoin.

La figure 2.2 est le premier écran que peut voir l'utilisateur. Il permet en utilisant les menus de lancer le jeu, de voir les meileurs scores, de quitter ou encore de consulter l'aide relative à ce jeu.

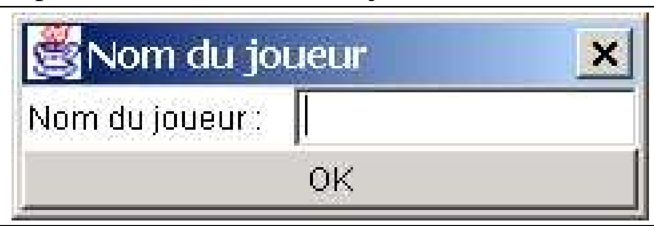


Figure 2.1 l'écran de démarrage



Si l'utilisateur choisit de jouer, alors le programme lui demande son nom au travers de l'écran de la figure 2.2

Figure 2.2 l'écran pour demander le nom du joueur



Un fois saisit le nom du joueur le jeu démarre véritablement avec le maise en place de l'écran de jeu de la figure2.2.

Figure 2.3 l'écran de jeu proprement dit



Conformément au cahier des charges, sur cet écran nous pouvons apercevoir les différents éléments associés au jeu.

3
Classes et objets

Ce premier chapitre décrit...

3.1 Objets et références

Un objet est caractérisé par :

- Une *identité*, qui assure son unicité.
- Un *état*, représenté, à un instant donné, par les valeurs des attributs qui le caractérisent.
- Un *comportement*, constitué par l'ensemble des actions, définies par des méthodes, permettant de consulter et modifier l'état de l'objet. Ces méthodes représentent *a priori* le seul moyen de manipuler l'état de l'objet : c'est le principe d'encapsulation, dont nous avons parlé dans le chapitre précédent.

3.1.1 L'état d'un objet

Considérons, par exemple, le héros des Tortues Java. Son état du moment est supposé être le suivant :



Nom du joueur :	momo54
==================	
Points de Vie :	100/100
Expérience :	42
Force :	42

Arme Portée :	Massue
Armure Portée :	ArmureAirain
Bouclier Porté :	BouclierBois
Contenu du sac	

C'est la machine virtuelle Java qui crée tous les objets, en fonction des requêtes du programme des Tortues Java, et qui leur attribue un numéro ou *clé* unique (disons, pour simplifier, 1 pour le premier objet, 2 pour le second, etc.), afin de pouvoir les référencer de manière univoque. La clé d'un objet ne changera jamais au cours de l'existence de l'objet.

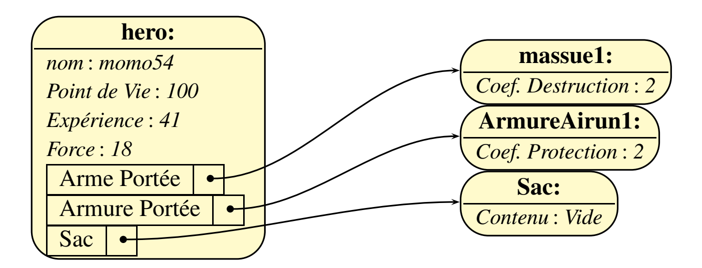
Convenons que les objets seront représentés dans la suite sous la forme graphique

suivante :

hero:
nom : *momo54*
Point de Vie : *100*
Expérience : *41*
Force : *18*
Arme Portée : *Massue*
Armure Portée : *ArmureAirain*
Sac : *vide*

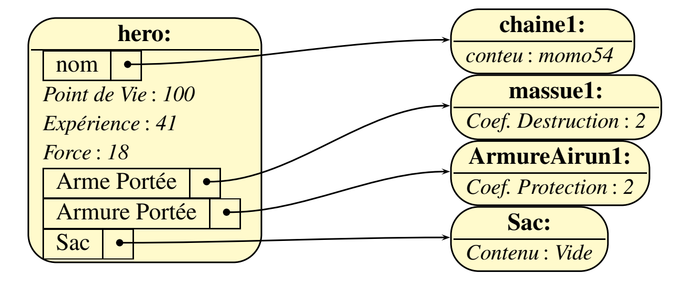
Cette représentation est évidemment assez naïve, le monde du programme des Tortues Java étant en fait constitué d'un réseau d'objets. En effet, si, dans le cahier des charges, le héros porte une massue, dans le « monde objet », il référence un objet représentant la massue, qui a ses propres identité, état et comportement. Il en est de même pour portée et le sac.

Une représentation plus fidèle de l'objet `héros` serait donc comme suit :



Si nous poussons ce raisonnement plus avant, `momo54` est logiquement un objet, de type « chaîne de caractères », dont la valeur est la chaîne `momo54`. En toute rigueur, il devrait également en être de même pour les entiers `100` et `42`. Toutefois, pour des questions de simplicité et d'efficacité, entre autres, les entiers Java sont considérés comme des objets de type primitif, qui ne sont pas soumis au principe d'encapsulation. Ils sont manipulés à la façon d'un langage de programmation n'utilisant pas d'objets, C par exemple, sans avoir besoin de recourir à des méthodes.

Voici donc une représentation plus exacte de notre objet :



Toutefois, dans un souci de simplification, les chaînes de caractères et les nombres seront désormais représentés comme des attributs primitifs, à la manière du second diagramme.

Évidemment, le héros n'est pas le seul objet de l'application. En particulier, il est à tout instant associé à une position sur la carte, qui peut être représentée de bien des façons, par exemple un couple de coordonnées *(x,y)*, ou encore une référence à

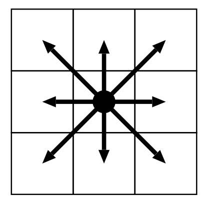


FIG. 3.1 – *Les déplacements possible du héros, à partir de la case centrale.*

un objet `case`. Une case peut elle-même être caractérisée par ses coordonées sur la carte, le paysage qui lui est liée, la présence éventuel d'un personnage, monstre ou héros, etc. Nous venons ainsi de mettre en évidence des éléments qui seront sûrement aussi des objets : la carte et les paysages. Déterminer comment ils seront représentés va mettre en évidence d'autres objets, et ainsi de suite.

Nous y reviendrons plus tard. Pour l'instant, seul le héros, avec sa massue et son armure, nous intéresse.

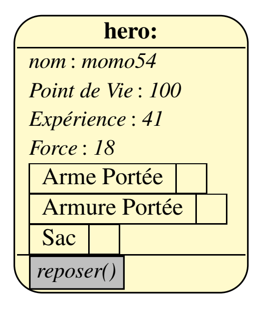
3.1.2 Le comportement d'un objet

Comme nous l'avons dit pour commencer, chaque objet est doté d'une indentité, d'un état, et aussi d'un comportement. Tâchons donc de définir une première ébauche du comportement du héros à partir du cahier des charges.

- Le héros peut se reposer, pour augmenter son capital de points de vie.

L'objet `heros` doit donc être doté d'une première opération, ou plutôt, selon le vocabulaire spécifique de Java, d'une *méthode*, sans paramètre, notée `reposer()`. Donc, lorsque cette méthode est invoquée, elle augmente le nombre de points de vie du héros.

Complétons la représentation graphique du héros :



- Le héros peut se déplacer dans une des cases adjacentes à celle où il se trouve, c'est-à-dire selon huit directions possibles, comme l'indique la figure 3.1.

À l'évidence, la méthode correspondante, `deplacer`, prend la direction du déplacement en paramètre.

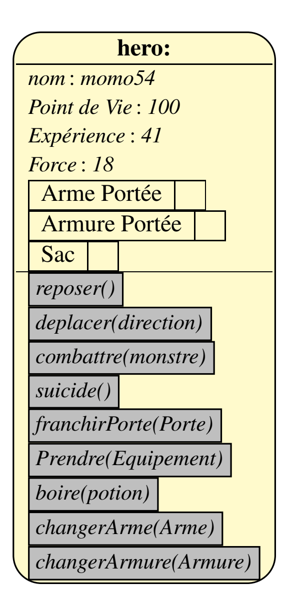
En outre, si le héros gagne une case déjà occupée par un monstre, ce dernier l'attaque et il y a donc combat (cf. le cahier des charges). Ce faisant, nous venons

de mettre en évidence une autre méthode, `combattre`, nécessaire pour modéliser cette partie du comportement du héros. Elle doit prendre en paramètre le monstre à combattre, puisqu'il faut être deux pour se battre !

Un examen détaillé du cahier des charges permet ainsi de déterminer progressivement les différentes méthodes nécessaires. Passons plus rapidement sur les restantes :

- Le héros peut se suicider, mettant ainsi fin au jeu (méthode `suicide()`).
- Le héros peut franchir la porte menant à la carte suivante, à condition de résider sur la case contenant la porte (méthode `franchirPorte()`).
- Le héros peut ramasser un des objets se trouvant sur la case où il réside (méthode `prendre(equipement)`).
- Le héros peut boire une potion contenue dans son sac (méthode `boire(potion)`).
- Le héros peut changer d'arme (méthode `changerArme(arme)`) ou d'armure (méthode `changerArmure(armure)`).

La figure suivante montre une représentation complète de l'objet `heros`.



, muni des méthodes appartenant *l'interface* de l'objet. Toujours selon le principe d'encapsulation, un utilisateur de l'objet ne peut « manipuler », c'est-à-dire modifier son état, qu'en utilisant l'une d'entre elles.

Si l'on applique, le principe d'encapsulation, seules les opérations définies dans le comportement d'un objet peuvent accéder à l'état de celui-ci. Pour l'instant, la seule chose que je peux demander à mon objet "héros" c'est de se reposer.



La notion d'objet n'est liée à aucun langage objet en particulier : c'est une façon de modéliser un univers. Nous avons abordé les notions fondamentales d'identité, d'état et de comportement d'un objet, ainsi que le principe d'encapulation. Êtes-vous capables d'expliquer chacune de ces notions ?

- Définition d'un commentaire. Tout texte apparaissant derrière `//+ est un commentaire`
- définition de la classe `Mauhly`
- Cette section définit la structure de l'état des objets créés. Les variables ainsi définies sont appelées *variables d'instances*. Chaque instance de `Mauhly` *i.e.*, chaque objet créé en utilisant cette classe comme modèle, aura donc son état structuré selon cette définition.
- Cette section définit le comportement des objets de classe `Mauhly` *i.e.* créé en utilisant cette classe comme modèle. Chaque opération définie dans cette section est appelée *méthode d'instance*, c'est à dire une méthode pouvant être appelée sur un objet de classe `Mauhly`.
- fin de la définition de la classe `Mauhly`.

```
// definition de la classe Maulhy
class Mauhly {
    private String nom;
    private int force;
    private int vie;
    private int exp;

    public int getForce() {
      return Force ;
    }
    public int getExp() {
      return exp ;
    }
    public int getVie() {
      return vie ;
    }
    private void setVie(int viep) {
      vie=viep;
    }
    public void reposer() {
      setVie(getVie()+10);
    }
    //<...>
} // class Mauhly
```

FIG. 3.2 – *La classe "Mauhly": vue générale*

3.2 Les Classes

Les objets "monstre" sont assez similaires à l'objet héros. Les monstres ont le même état, la même interface de comportement, mais à la différence du héros, ils sont autonomes, ils décident d'eux-mêmes si ils doivent se déplacer, boire un potion, ou changer d'armes.

Tous les monstres de type "maulhy" appartiennent à la même classe. La structure de leur état est la même, le comportement de chaque "maulhy" est indentique.

Dans un langage de classe comme Java, C++, Eiffel, avant de créer les objets, il faut définir les classes de ces objets. Ces classes contiennent la définition de l'état et du comportement. Nous allons définir progressivement, la classe "Maulhy" qui va permettre de créer les objets "maulhy1", "maulhy2", ...

En java, la définition des classes est contenue dans des fichiers avec l'extension ".java". Il est possible de définir plusieurs classes dans le même fichier, mais nous nous conformerons aux conventions d'écriture et de ce fait nous ne déclarerons qu'un classe par fichier. Le fichier porte le nom exact de classe. La classe "Maulhy" sera donc définie dans le fichier "Maulhy.java" dont le contenu est décrit dans la figure 3.4

```
class Mauhly {
    // ...
    private int force;
    public string force ; // erreur. force est deja utilise

    public int force() { // ok, c'est une methode
      return Force ;
    }
    public void force() { // erreur, force() existe deja
      // ...
    }
} // class Mauhly
```

FIG. 3.3 – *Conflits de nommage dans une classe*

La classe `Mauhly` doit être considérée comme une usine à fabriquer des objets de classe `Maulhy`; on dit des *instances* de `Mauhly`. Je peux donc créer autant d'instances de la classe `Mauhly` que je veux; tous auront le même comportement tel qu'il est définit dans la classe et chacun aura ses propres valeurs pour ses variables d'instances.

Supposons que l'on cée 3 trois instances de la classe Maulhy que la machine Java attribue les indentité de manière naïve, nous pouvons obtenir trois objets:

Maulhy1:	**Maulhy2:**	**Maulhy3:**
nom : Pascal	*nom : Gérald*	*nom : Laurent*
Point de Vie : 100	*Point de Vie : 200*	*Point de Vie : 200*
Expérience : 41	*Expérience : 4100*	*Expérience : 410*
Force : 18	*Force : 21*	*Force : 21*

La classe `Mauhly` définit un espace de nommage. Deux variables d'instance ne peuvent porter le même nom au sein de la même classe. Il en est de même pour les méthodes.Par contre une méthode et une variable d'instance peuvent porter le même nom (voir figure 3.2)

Enfin les conventions de nommage sont les suivantes :

- Un nom de classe commence par une majuscule. Si le nom est un nom composé, on écrit le mot sont sans tirets, chaque début de mot prend une majuscule. par exemple, imaginons que nous ayons besoin d'une classe "Bouclier de fer", la classe s'appelera `BouclierFer`.
- Un nom de variable d'instance commence par une minuscule. Si le mot est composé, chaque mot suivant prend une majuscule. Par exemple, supposons que nous ayons besoins d'une variable "dégat de l'arme", nous pourrons l'appeler `degatArme`
- Un nom de méthode est soumis au même convention qu'un nom de variable d'instance. Il n'y a pas d'ambiguité; les parenthèses suivent toujours un nom de méthode.
- il est possible de déclarer plusieurs instructions par ligne. Par exemple:

```
private int force; private int vie; private int exp;
```

définition du contrôle d'accès: Une variable d'instance peut-être déclarée `public` ou `private`. Cela signifie ici, que seules les instances de la classe `Mauhly` peuvent accéder à cette variable.

Chaque variable d'instance est typée. Cette variable est de type `int` pour entier, elle n'accepte donc que des entiers pour valeur.

Toute variable d'instance peut déclarer une valeur initiale. A la création des objets, si cette valeur n'est pas modifiée par d'autres mécanisme `exp` de l'objet créé vaudra 0.

Chaque méthode d'instance est aussi soumise au contrôle d'accès. `public` signifie que n'importe qui *i.e.* n'importe quel objet peut appeler cette méthode.

```
class Mauhly {
   private String nom;
   private int force;
   private int vie;
   private int exp=0;

   public int getForce() {
     return force ;
   }
   public int getExp() {
     return exp ;
   }
   public int getVie() {
     return vie ;
   }

   private void setVie(int viep) {
     vie=viep;
   }


   public void reposer() {
     setVie(getVie()+10);
   }
   // <...>
} // class Mauhly
```

FIG. 3.4 – *la classe Maulhy, types et contrôle d'accès*

Par convention et pour améliorer la lecture, on se borne à une instruction (terminée par un `;` par ligne.

– Il est possible de déclarer les méthodes puis les variables d'instances, ou même de mixer les deux, par convention on déclare d'abord tout ce qui est relatif à l'état, puis ce qui est relatif au comportement.

Les contrôleurs d'accès comme `public` ou `private` s'appliquent aux variables et aux méthodes d'instances. Ils sont le mécanisme permettant de mettre en place le principe d'encapsulation dans le langage java.

Si nous voulons nous conformer au principe d'encpasulation, il faut définir les variables d'instances qui caractérise l'état de l'objet en `private` et les méthodes qui caractérise son comportement comme `public`. Toutes les méthodes d'instances n'ont pas à être publiques, certaine peuvent être réservée pour usage interne et donc être déclarée comme `private`.

Java est un langage typé. Toute définition de variable doit être typée. L'objectif essentiel de cette définition est de faire du contrôle de type. C'est une des fonctions

du compilateur. Quand nous allons compiler ce programme afin de générer un programme exécutable, le compilateur va bien sur vérifier la syntaxe du programme mais aussi la cohérence des types. Si il détecte par exemple lors d'une affectation qu'un nombre réél est affecté à un entier, il va signaler cette incompatibilité de type.

Nous verrons plus loin que la cohérence des types est un élément fondamental du langage.

Il faut toutefois être attentif à la manière dont est réalisé l'encapsulation. Le principe d'encapsulation dit que pour accéder à l'état d'un objet il faut absolument passer par les méthodes de son interface. Ici, l'encapsulation réalisé en java à un sens un peu différent. En effet, la variable d'instance `force` d'une instance de `Mauhly` sera accessible non seulement à cet objet mais aussi par tous les objets de même classe. C'est une encapsulation basée sur les classe.

D'autres langages objets comme Smalltalk, prennent une autre stratégie et propose une encapsulation basée sur les objets. Seul l'objet lui-même peut accéder à son état.

Une méthode est basiquement une opération qui prend des paramètres, accède à l'état de l'objet et renvoie un résultat. Java est un langage typé, les paramètres sont donc typés et le résultat aussi. La forme générale de toute déclaration de méthode est la suivante:

```
<controleur d'acces> <type retour> <nom methode>([liste parametres formels]*) {
  // body
}
```

La *portée* d'un paramètre formel se limite au corps de la méthode dans lequel il est déclarer. Cela signifie que `viep` n'a de sens que dans le corps de `setVie`. La notion de portée des identificateur est très importante dans les langages informatique en général. La portée d'un nom de variable d'instance est limité au corps de classe. La portée d'un nom de classe est limité au programme dans lequel il est utilisé. Ce qui signifie que pour un programme, il ne peut exister deux classe portant le même nom.

La méthode `reposer` est intéressante: cette méthode s'appuie sur des méthodes "organiques" que tout programmeur java s'attend à trouver dans la définition d'une classe. Le reflexe est le suivant: pour toute variable d'instance, je définit une méthode d'accès et une méthode de modification. La variable d'instance est déclarée `private`, l'accesseur et le modificateur `public` ou `private`, selon les besoins. Ici, n'importe quel objet peut savoir combien de vie a un objet de classe `Maulhy` mais seule une instance de `Mauhly` peut modifier cette valeur.

Par convention, les accesseurs sont toujours préfixés par `get` et le modificateur par `set`, mot anglais que nous n'avons pas préféré traduire.

Quand on appelle la méthode `reposer()` sur une instance de `Maulhy`, les instructions situées dans le corps de cette méthode sont executées. Pour pouvoir exécuter `setVie(getVie()+10)`, il faut d'abord exécuter `getVie()`. Cette méthode est appelée sur l'objet lui-même, c'est donc bien la méthode `getVie()` que nous avons définie que nous allons appeler. Cette méthode renvoie la valeur associée à la variable

☐ Chaque méthode définit le comportement exact correspodant à son action dans le *corps* de la méthode situé entre les accolades. Quand la méthode est appelée, c'est ce code qui est exécuté.

☐ Les variables d'instances peuvent être accédées dans le corps des méthodes. Ici, nous affectons 10 à la variable `force`. 10 est bien un entier, l'affectation est donc possible.

☐ Les méthodes d'instances renvoie toujours une valeur de retour sanf pour le constructeur, que nous verrons juste après. Ici `int` est le type de la valeur de retour, la valeur retournée est spécidiée dans le corps de la méthode par le mot-clef `return`. La valeur retournée doit avoir un type compatible avec le type de la valeur de retour.

☐ Le mot-clef `void` signifie que la méthode ne retourne rien.

☐ Les parenthèses signifient que cette méthode ne prend aucun paramètre.

```
class Mauhly {
  private String nom;
  private int force;
  private int vie;
  private int exp=0;

  public int getForce() {
    return force;
  }
  public int getExp() {
    return exp ;
  }
  public int getVie() {
    return vie ;
  }

  private void setVie(int viep) {
    vie=viep;
  }

  public void reposer() {
    setVie(getVie()+10);
  }
  // <...>
} // class Mauhly
```

FIG. 3.5 – *la classe "Maulhy": les méthodes*

d'instance `vie`. Disons qu'à cet instant cette variable vaut 10, `getVie()` renvoie donc 10. Il ne reste plus qu'à exécuter `setVie(10}10)`, soit `setVie(20)`. 20 forme ici ce qu'on appelle le *paramètre effectif* de la méthode `setVie`. Le *paramètre formel* de `setVie`, à savoir `viep` (comme vie-paramètre), va prendre la valeur du paramètre effectif, pour exécuter les instructions située dans le corps de la méthode `setVie`. Le type du paramètre effectif doit être compatible avec le type du paramètre formel, ce qui est bien le cas ici. Le corps de la méthode `setVie` peut-être executée maintenant. `vie=viep;` va être interprété comme `vie=20;`. La variable d'instance vient d'être modifiée. La méthode `setVie` termine son exécution, ce qui termine aussi l'exécution de la méthode `reposer`.

Il était possible d'écrire la méthode `reposer` de la manière suivante:

```
public void reposer() {
  vie=vie}10;
}
```

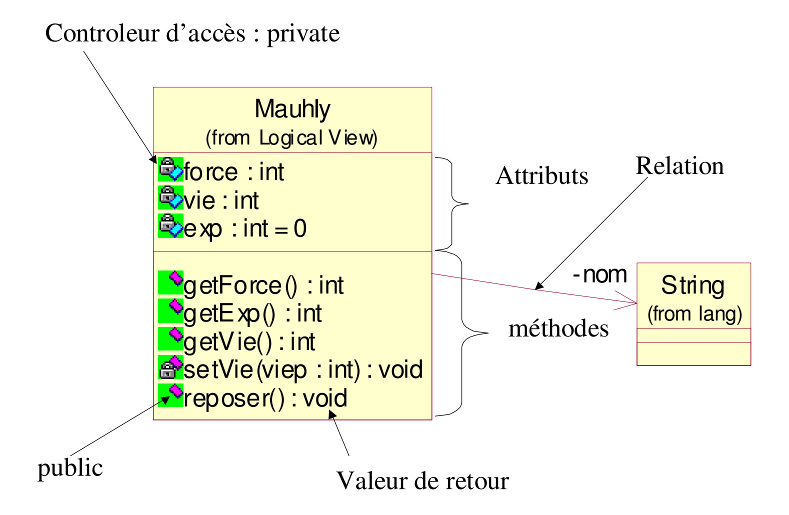


FIG. 3.6 – *La représentation UML de la class Mauhly*

Le comportement de ces deux définitions est exactement le même. Mais un programmeur objet préférera la première solution. En effet, dans ce cas, la définition de `reposer` ne dépend que des fonctions d'accès. Si par exemple je décide de changer le nom de la variable d'instance `vie`, seul les accésseurs et modificateurs sont concernés. Dans la deuxième solution, tous les utilisateurs de la variable d'instance sont concernés. L'impact d'un changement est potentiellement plus fort dans la deuxième solution.

Il y a une deuxième raison: si je décide que toute modification de la vie d'un "Maulhy" ne peut se faire que si il n'est pas paralysé par exemple, je peux rajouter ce test dans la méthode `setVie` sans modifié le reste de la définition de la classe. La encore, l'impact du changement est réduit en appliquant juste de convention d'écriture. Ces conventions ne viennent en fait que d'un respect plus grand du principe d'encapsulation.

Nous utiliserons assez souvent dans ce livre une représentation diagrammatique des classes. Cette représention est conforme à la norme UML. Cette représentation est généralement utilisée dans les phases d'analyse et de conception d'un logiciel. Elle permet entre autre de donner une vue synthétique de l'organisation statique et dynamique d'un logiciel favorisant ainsi la compréhension générale.

La classe `Mauhly` est représentée dans la figure **??**.

La figure 3.6 est un diagramme de classe montrant la classe `Mauhly`. On observe sur cette figure la différence entre type primitif et objet. Les types primitifs entier, réels, booléan, caractères ne sont pas des objets en Java. Il n'ont donc pas de classes définissant leurs états et comportement. Ils apparaissent comme des attributs simple en UML. Par contre, en Java, `String` est une classe, donc la variable d'instance `nom` est en fait une référence sur un objet de classe `String`. Mais je n'ai pas personnellement définit la classe `String`. Elle est en fait définie dans la bibliothèque Java. Comme cette bibliothèque est très grande, les classes sont rangées dans des sortes

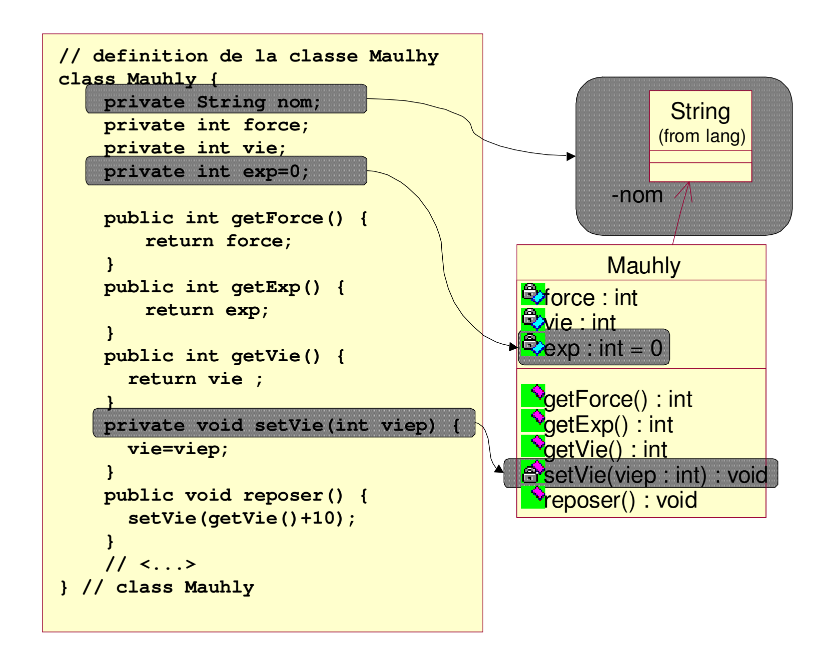


FIG. 3.7 – *Correspondances Java/UML*

de répertoires et sous-répertoires et forment des espaces de nommage. En Java, Ils sont appelés “package”. Nous verrons plus tard que les packages sont plus que juste des espaces de nommage pour classes, mais pour l’instant cette vision nous suffit. La classe `String` fait partie du package `java.util`. Elle ne s’appelle pas `String` mais `java.lang.String`. Des mécanismes que nous détaillerons ultérieurement nous permettent de l’appeler juste `String` dans notre cas.

La figure 3.7 illustre les correspondances entre java et UML. Elle montre comment une référence en java de la classe `Mauhly` de type `String` se matérialise en UML par une relation ici unidirectionnelle. Nous montrerons d’autre type de relations plus tard.

Dans cette section, nous avons étudié la définition d’une classe. Une classe contient la définition permettant d’instancier une multitude d’objet de même classe. Dans un langage de classe, les classes existent avant les objets. Pusieurs définitions, principe et éléments de vocabulaire sont importants:

- Variables et méthodes d’instance.
- Paramètres formels, paramètres effectifs, valeur de retour.
- la notion de portée des variables.
- la différence entre le paradigme d’encapsulation et comment ce principe est réalisé dans un langage objet. Ici par des controleur d’accès avec une encapsulation basée sur les classes et non sur les objets.
- la notion de langage typé, de cohérence des types, de contrôle de type.

3.3 Instanciation et appel de méthode

Dans les langages de classes comme Java, les classes existent avant les avant les objets. Comme, a priori pour exécuter un programme, il faut des méthodes et ces méthodes ne peuvent être appelées que sur un objet, comment faire pour créer le premier objet ?

Le problème a été résolu en introduisant une *méthode de classe*. Cette méthode n'a pas besoin qu'un objet existe pour pouvoir être appelée, elle s'applique directement à la classe. Le mot-clef `static` permet de déclarer une méthode comme méthode de classe et non d'instance.

Dans tout programme java, une méthode de classe particulière appelée `main` constitue le point d'entrée du programme. Ce méthode doit avoir impérativement le *profil* suivant:

```
public static void main(String args []) {
  // ...
}
```

Le paramètre formel `String args[]` est un tableau de chaine de caractères. Nous verrons la manipulation des tableau un peu plus loin. Ce tableau permet récupérer les paramètres de la ligne de commande. Par exemple, si l'utilisateur lance un programme Java en tapant cette commande:

```
% cd Tortues
% java Copy Maulhy.java Masini.java
```

Alors, à l'exécution, le paramètre formel `args` aura pour valeur un tableau de chaînes de caractères contenant `Maulhy.java` et `Masini.java`.

Nous allons maintenant créer véritablement des objets de classe `Maulhy`. Pour cela il me faut un point d'entrée; soit j'ajoute une méthode `main` à la classe `Maulhy`, soit je crée une nouvelle classe, juste pour héberger le point d'entrée. J'opte pour la seconde solution, juste par convention de programmation. Je crée donc une nouvelle classe, que j'appelle arbitrairement `Main` pour héberger le point d'entrée.

Le programme de la figure 3.8 instancie un objet de classe `Mauhly` et effectue plusieurs appel de méthodes. Le résultat de l'exécution de ce programme apparait dans la figure 3.3. Intuitivement ce résultat semble correct. En effet, la méthode `main` commence par créer un objet de classe `Mauhly`, imprime le nombre de point de vie de cet objet, demande au monstre de se reposer puis affiche le nombre de points de vie à nouveau.

L'exécution de la méthode de classe `main` de la classe `Main` (cf 3.8) commence par créer un objet de classe `Mauhly` et garde une référence sur cet objet nommée `momo`.

Il nous faut donc définir maintenant ce qu'est une référence dans un langage de classe et après nous regarderons de plus près les subtilités liées à la création de notre premier objet.

D'abord, il faut se rendre compte que Si `main` ne garde pas de référence sur le nouvel objet, il lui sera impossible par la suite de le désigner. En effet, l'identité

🄋 définition d'une variable locale. Il s'agit ici d'un référence sur un objet de classe `Mauhly`. La portée de cette référence est limité au corps de la méthode `main`.

🄋 `new` crée véritablement un objet de classe `Mauhly`.

🄋 Voici un appel de méthode ou encore un envoi de message. L'*objet référencé* par la variable `momo` est appelé le *receveur*. la méthode `reposer()` est appelé sur l'objet receveur.

🄋 `System.out.println(...)` permet d'imprimer le paramètre effectif sur la sortie standard. Ce paramètre doit être de classe `String`.

```
class Mauhly {
    private String nom;
    private int force;
    private int vie;
    private int exp=0;
    public int getForce() { return force; }
    public int getExp() { return exp; }
    public int getVie() { return vie ;}
    Private void setVie(int viep) { vie=viep; }
    public void reposer() {
      setVie(getVie()+10);
    }
}
class Main {
  public static void main(String args []) {
    Mauhly momo=new Mauhly();
    System.out. println (momo.getVie());
    momo.reposer();
    System.out.println(momo.getVie());
  }
}
```

FIG. 3.8 – *Instanciation*

```
bar ex1 55 % cd ex1
bar ex1 5- % javac *.java
bar ex1 57 % java Main
0
10
bar ex1 58 %
```

FIG. 3.9 – *résultat du programme de la figure 3.8*

de cet objet n'est reservé qu'à l'usage exclusif de la machine virtuelle java. Donc le seul moyen pour le désigner et de lui attribuer une référence dès sa naissance. si cette précaution n'est pas prise, ce n'est pas très grave, juste inutile, car l'objet nouvellement créé est alors définitivement perdu car non-atteignable.

Une référence est un lien typé sur un objet typé. Les types doivent être compatible. Bien sur, deux types identiques sont compatibles.

Pour toute référence, il y a donc deux types : le type de la référence appelé *type statique* et le type de l'objet référencé appelé *type dynamique*. La compréhension de la notion de type statique/type dynamique est fondamentale pour la suite.

une référence n'a que deux états possibles; soit elle désigne un objet dont le type est compatible avec le type de la référence, soit elle désigne rien et dans ce cas sa valeur est égale à `null`. `null` est un mot-clef du langage java, il permet d'initialiser

```
class Main {
    public static void main(String args[]) {
àlonglabelémomo1èé
       Mauhly momo=new Mauhly();
è
    }
}
```

⇕

```
class Main {
    public static void main(String args[]) {
àlonglabelémomo2èé
       Mauhly momo=null;
       momo=new Maulhy();
è
    }
}
```

⇕

```
class Main {
    public static void main(String args[]) {
àlonglabelémomo3èé
       Mauhly momo;
       momo=new Maulhy();
è
    }
}
```

FIG. 3.10 – *Références*

les références. La figure 3.10 montre plusieurs écritures équivalentes pour la création d'une référence et son initialisation.

momo en plus d'être une référence est une variable locale à la méthode main. En effet, elle est déclarée à l'intérieur du corps de cette dernière. Contrairement à une variable d'instance, la portée d'un variable locale est limitée au corps de la méthode dans laquelle elle est définie. la variable locale momo est rééllement créée quand la méthode main est appelée et disparait à la fin de l'exécution de cette dernière.

Si la référence "momo" est une variable locale, elle ne fait partie de l'état d'aucun objet. Pourtant cette variable doit forcément exister quelque part . . . .

Elle existe en fait dans le contexte d'appel de méthode main, lui-même faisant partie de la pile d'appel du programme. Les objet eux sont en fait créés dans une

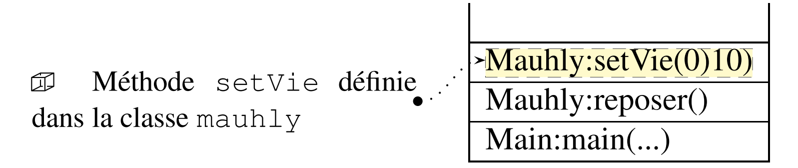


FIG. 3.11 – *Pile d'appel*

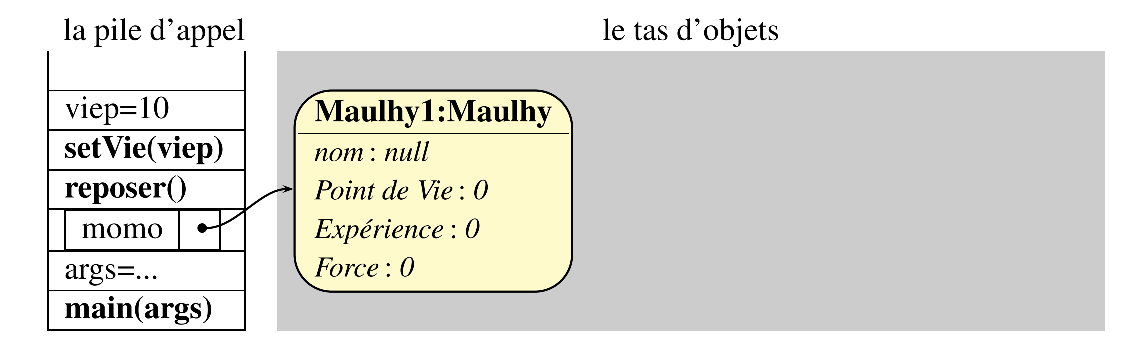


FIG. 3.12 – *la pile d'appel*

espace appelé le *tas*.

Avec un peu de recul, on voit bien qu'exécuter un programme objet, c'est appeler une méthode qui en s'exécutant va appeller d'autre méthodes et ainsi de suite, jusqu'à atteindre la dernière ligne de la méthode `main`.

Par exemple, lors de l'exécution de la méthode `setVie(...)`, appelée par la méthode méthode d'instance `reposer()`, elle même appelée par la méthode de classe `main`. il y a donc trois méthodes empilées: `reposer()` attend la fin de l'exécution de `setVie()` et `main` attend la fin de l'exécution de `reposer`. La figure 3.11 représente l'état de la pile d'appel lorsque le pointeur d'exécution est sur la ligne 19 (cf3.8).

Chaque méthode possède un contexte d'exécution formé de ses paramètres effectifs et de ses variables locales. La figure **??** représente la pile d'appel avec les variables locales et la paramètres effectifs.

Cette pile d'appel peut être visualisée en utilisant le débogueur java: `jdb`. Pour l'utiliser, il faut péalablement compiler les fichiers java avec l'option de compilation `-g`.

```
bar instanc 67 % cd ex1/
bar ex1 68 % ls
Mauhly.java
Main.java
bar ex1 69 % javac -g *.java
```

On peut ensuite lancer la machine virtuelle Java en mode debug en appelant `jdb`.

```
bar ex1 70 % jdb Main
Initializing jdb...
```

```
0xb0:class(Main)
>
```

Pour visualiser la pile d'appel à l'entrée de la méthode `setVie(...)`, je dois demander à la machine virtuelle de s'arréter quand elle commmence à exécuter cette méthode.

```
bar ex1 70 % jdb Main
Initializing jdb...
0xb0:class(Main)
> stop in Mauhly.setVie
Breakpoint set in Mauhly.setVie
>
```

Je demande maintenant à la machine d'exécuter le programme

```
bar ex1 70 % jdb Main
Initializing jdb...
0xb0:class(Main)
> stop in Mauhly.setVie
Breakpoint set in Mauhly.setVie
> run
run Main
0running ...
main[1]

Breakpoint hit: Mauhly.setVie (Mauhly:16)
main[1]
```

La machine s'arrête au point d'arrêt que je lui ai spécifié. je peux maintenant lui demander d'afficher la pile d'appel

```
...
> run
run Main
0running ...
main[1]

Breakpoint hit: Mauhly.setVie (Mauhly:16)
main[1]  where
  [1] Mauhly.setVie (Mauhly:16)
  [2] Mauhly.reposer (Mauhly:19)
  [3] Main.main (Main:5)
main[1]
```

On observe bien qu'à ce moment là, la méthode `main` a appelé la méthode `reposer` qui a elle même appelée la méthode `setVie`.

je demande maintenant à la machine d'afficher le contexte d'exécution de la méthode `setVie(...)`.

```
...
> run
run Main
0running ...
main[1]
Breakpoint hit: Mauhly.setVie (Mauhly:16)
main[1]  where
  [1] Mauhly.setVie (Mauhly:16)
  [2] Mauhly.reposer (Mauhly:19)
  [3] Main.main (Main:5)
main[1] locals
Method arguments:
Local variables:
  this = Mauhly@3dc15b79
  viep = 10
```

On observe ici, la valeur du paramètre formel `viep` et une référence appelée `this`. Rappelez-vous, pour exécuter un méthode, il faut un objet receveur. Doc si `setVie(...)` s''exécute, elle s'exécute sur un objet. `this` est une référence sur l'objet receveur du message. Nous reviendrons plus tard sur l'utilisation de cette référence.

Il est possible d'inspecter la pile d'appel en parcourant les différents blocks de la pile d'appel et ainsi d'afficher les contexte d'exécution de chaque méthode.

```
...
main[1] up
main[2] where
  [2] Mauhly.reposer (Mauhly:19)
  [3] Main.main (Main:5)
main[2] locals
Method arguments:
Local variables:
  this = Mauhly@3dc15b79
main[2] up
main[3] where
  [3] Main.main (Main:5)
main[3] locals
Method arguments:
Local variables:
```

```
  args =
  momo = Mauhly@3dc15b79
main[3]
```

Dans le contexte d'exécution j'ai accès à la référence `momo`, je peux demander d'afficher la valeur de l'objet référencé.

```
...
main[3] where
  [3] Main.main (Main:5)
main[3] locals
Method arguments:
Local variables:
  args =
  momo = Mauhly@3dc15b79
main[3] dump momo
momo = (Mauhly)0xe1 {
    private int exp = 0
    private int vie = 0
    private int force = 0
    private java.lang.String nom = null
}
main[3]
```

On peut tout à fait vérifier que je n'ai pas raconté de conneries jusqu'à maintenant[1]. Je demande maintenant à la machine de terminer l'exécution. je remonte en haut de la pile et je lui demande de continuer.

```
...
main[3] dump momo
momo = (Mauhly)0xe1 {
    private int exp = 0
    private int vie = 0
    private int force = 0
    private java.lang.String nom = null
}
main[3] down
main[2] down
main[1] cont
0
10

Current thread "main" died. Execution continuing...
```

1. et aussi vérifier le serieux des relecteurs ;-)

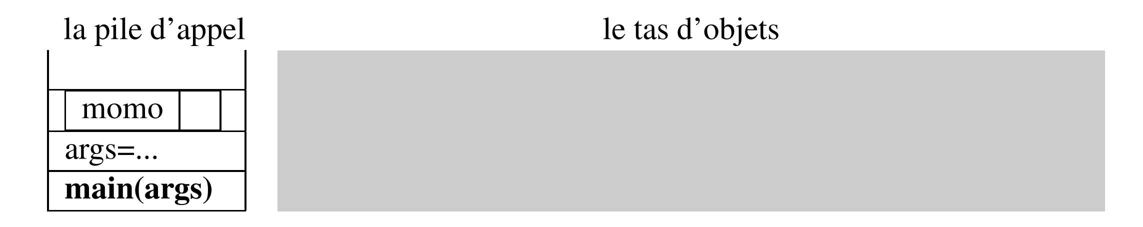


FIG. 3.13 – *public static void main(...)*

```
>
Main exited
```

Quand la fin de la méthode `main` est atteinte, le programme est terminé et la machine virtuelle java s'arrête. Je vous engage faire cette manipulation vous-même sur votre machine même si `jdb` est un débogueur un peu rustre, il est suffisant pour déboguer des petits programmes exemples. Pour de plus amples exmplication sur le fonctionnement du débogueur, je vous renvoie à la documentation fournie avec le jdk (section "tools").

Nous pouvons maintenant exécuter notre programme au pas à pas en utilisant cette visualisation graphique de la pile et dus tas. Vous pouvez faire exactement la même chose sur votre machine en utilisant un débogueur et vérifier que notre représentation graphique des choses est correcte.

1. L'utilisateur lance la machine java. La pile et le tas sont vide pour l'instant [2].
2. L'exécution démarre sur la méthode `main(...)`. Donc la méthode `main` avec son contexte d'exécution est empilée dans la pile (cf figure 2).
   Il faut noter que dans cet état la variable `momo` est non initialisée, elle est donc pour l'instant à `null`.
3. On exécute maintenant la première ligne.

```
  public static void main(String args []) {
=>  Mauhly momo=new Mauhly();
    System.out. println (momo.getVie());
    momo.reposer();
    System.out. println (momo.getVie());
  }
```

   Pour affecter une valeur à la référence `momo`, il faut déjà évaluer l'expression de droite `new Mauhly()`. Cette instruction crée véritablement l'objet dans le tas. L'état des différentes variables est soit celui spécifié dans la classe comme pour `int exp=0`, sinon c'est le compilateur qui attribue des valeurs par défaut; 0 pour les entier, `null` pour les référérences.

2. Ce qui n'est pas totalement vrai en ce qui concerne le tas. En effet, toutes les variables de classes sont initialisées dès le chargement des classes (cf section **??**). Par exemple, la classe `System` est chargée systèmatiquement dès le lancement de la machine Java, et `System.out` est bien une référence sur un objet de classe `PrintStream` qui existe dans le tas

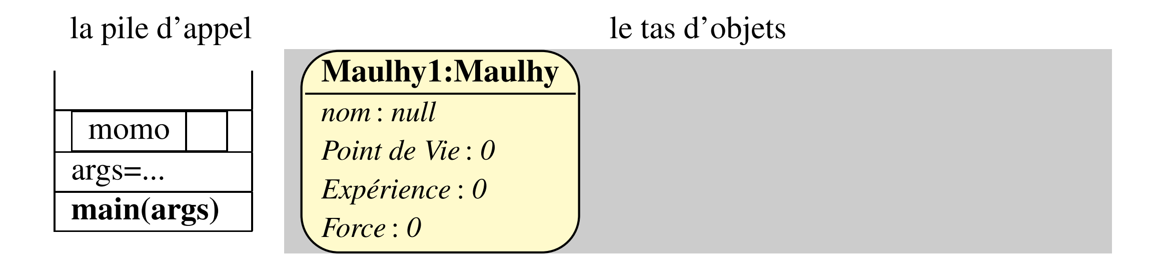


FIG. 3.14 – *new Mauhly()*

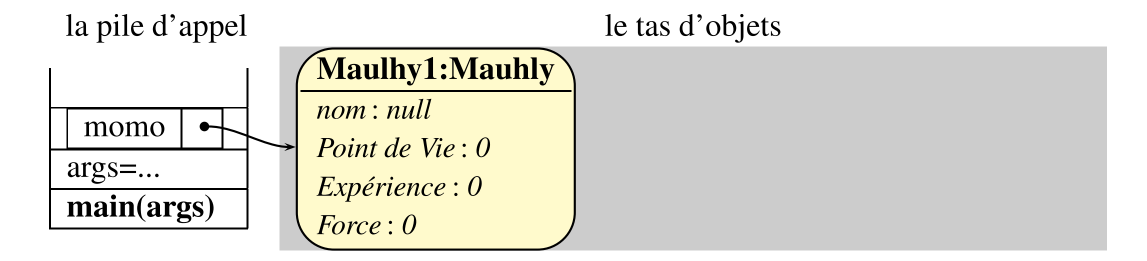


FIG. 3.15 – *Mauhly momo=new Maulhy()*

Donc, l'évaluation de l'expression `new Mauhly()` génère un nouvel état de la machine virtuelle (cf figure 3)

le résultat de l'exécution de `new Mauhly()` est une référence sur l'objet nouvellement créé. Comme l'affectation ver `momo` n'est pas encore réalisée, la référence `momo` reste à `null`.

4. On réalise maintenant l'affectation `Mauhly momo=new Maulhy()`, ce qui a pour effet de mettre à jour la référence `momo` (cf figure 5).
5. nous continuons l'exécution sur jusqu'à la troisième ligne. L'exécution de `System.out.println(momo.getVie())` a pour effet d'afficher 0, la valeur initiale de notre objet nouvellement créé.

```
  public static void main(String args[]) {
    Mauhly momo=new Mauhly();
    System.out. println (momo.getVie());
=> momo.reposer();
    System.out. println (momo.getVie());
  }
```

L'objet référencé par `momo` recoit le message `reposer`. Il faut donc empiler la méthode `reposer()` (cf figure 6).

Le code a exécuter est maintenant le suivant:

```
class Mauhly {
    // ....
    public int getVie() {
      return vie ;
    }
    private void setVie(int viep) {
```

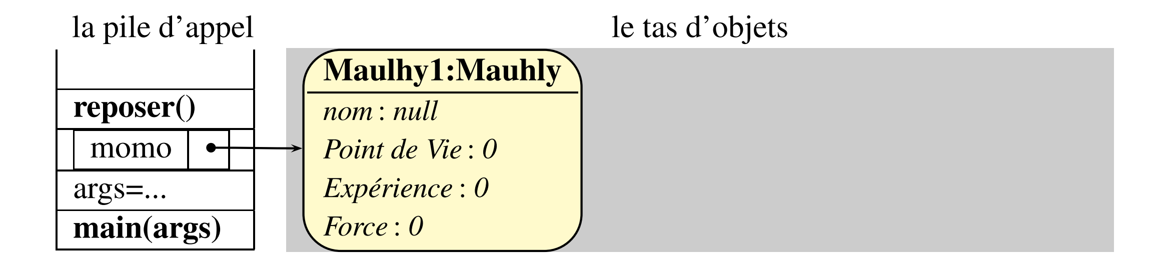


FIG. 3.16 – *Mauhly momo=new Maulhy()*

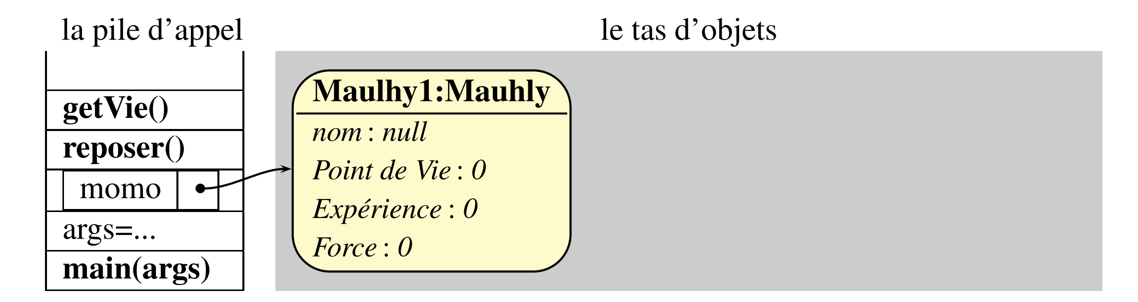


FIG. 3.17 – *Mauhly momo=new Maulhy()*

```
        vie=viep;
    }
    public void reposer () {
        setVie(getVie()+10);
    }
}
```

Il faut commencer par évaluer la partie entre les parenthèses `getVie()+10` et commencer par appeler la méthode `getVie` qui va retourner 0 (cf figure **??**)
.
Il ne reste plus qu'a exécuter la méthode `setVie(0)10)`. On empile la méthode `setVie(...)`.

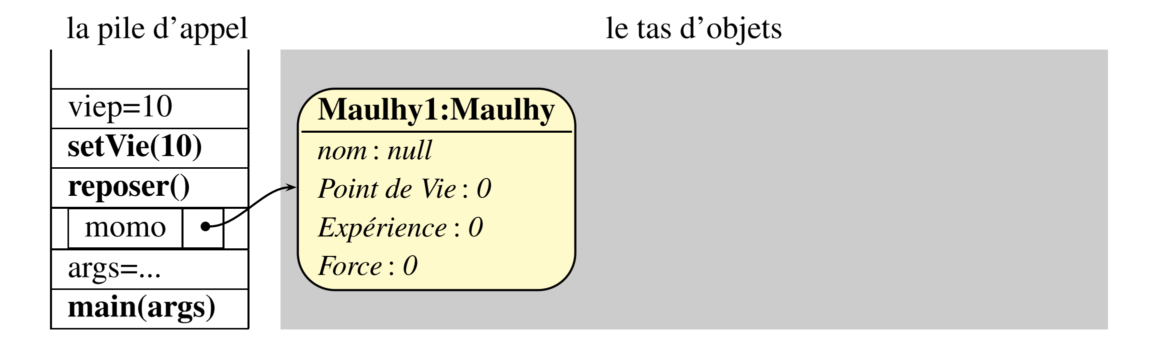


FIG. 3.18 – *Mauhly momo=new Maulhy()*

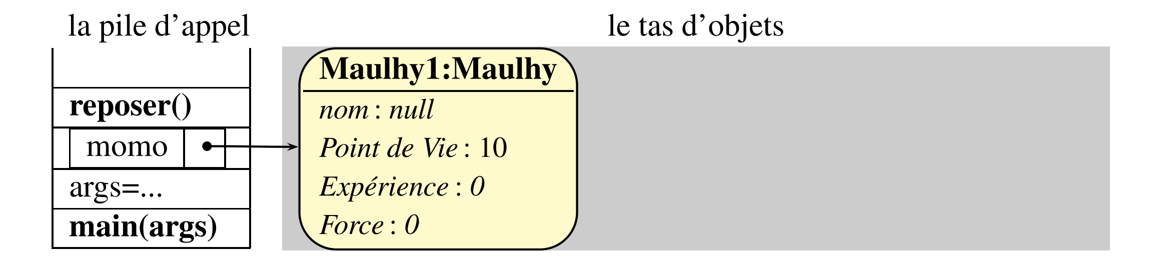


FIG. 3.19 – *Mauhly momo=new Maulhy()*

6. L'exécution de la méthode `setVie()` a pour effet de modifier l'état de l'objet `mauhly1` (cf figure **??**).

3.4 Instanciation et constructeur

Jusqu'à présent, l'état initial de l'objet est fonction des valeurs par défaut du compilateur et expressions d'initialisation fournies dans la définition de la classe.

Supposons que je veux maintenant fixer l'état initial d'un objet en fournissant des paramètres lors de sa construction. Par exemple, j'ai envie de créer un monstre de force égale à 10, de vie égale à 100 et de nom "affreux".

Bien évidemment, je peux créer un objet monstre et fixer ensuite chacune de ses caractéristiques en passant par les modificateurs i.e. `setVie(...)`,`setExp(...)`.

Le programme principal ressemble alors à:

```
public static void main(String args []) {
  Mauhly momo=new Mauhly();
  momo.setVie(100);
  momo.setForce(10);
  momo.setNom("affreux");
  // ...
}
```

En définissant une méthode particulière dans la classe `Maulhy`, je peux créer et initialiser mon objet de la manière suivante:

```
public static void main(String args []) {
  Mauhly momo=new Mauhly("affreux",100,10);
  // ...
}
```

Pour que cela fonctionne, la classe `Maulhy` doit définir une nouvelle méthode appelée `constructeur`.

Il est facile de penser que le constructeur construit l'objet. En fait il n'en est rien, lors de l'exécution du constructeur, l'objet existe déjà. Le constructeur peut être considéré comme une méthode d'instance particulière qui va fixer l'état initial de l'objet.

Reprenons notre programme principal:

Les expressions d'initialisation sont exécutées avant l'exécution du constructeur. Cela signifie que lorsqu'on exécute le code du constructeur de `Mauhly`, la valeur de `exp` est déjà à 0 [3].

Un constructeur a deux caractéristiques essentielles: Il porte exactement le même nom que la classe, aucune valeur de retour n'est déclarée.

Lorsqu'on exécute le constructeur, il faut bien se rendre compte que l'objet est déjà construit. L'exécution du constructeur va juste permettre de fixer son état initial. Il est donc possible dans un constructeur d'appeller des méthodes d'instances.

```
class Mauhly {
    private String nom;
    private int force;
    private int vie;
    private int exp=0;

    public Mauhly(String nomp, int forcep, int viep) {
        nom=nomp;
        force=forcep;
        setVie(viep);
    }
    // ...
    public int getVie() {
      return vie ;
    }
    private void setVie(int viep) {
      vie=viep;
    }
    public void reposer() {
      setVie(getVie()+10);
    }
}
```

FIG. 3.20 – *Contructeur*

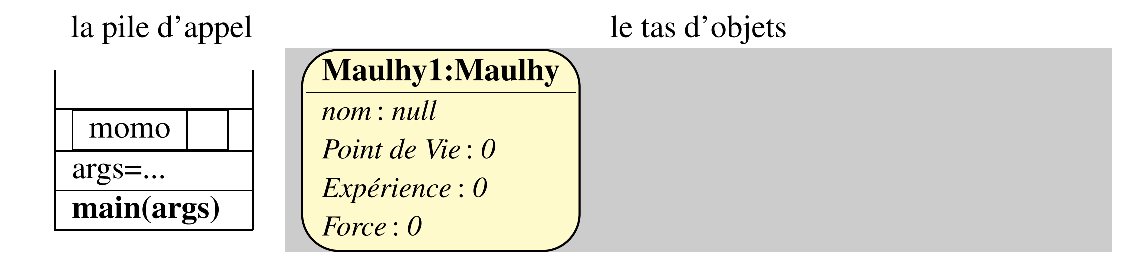


FIG. 3.21 – *new Mauhly("affreux",100,10)*

```
public static void main(String args[]) {
  Mauhly momo=new Mauhly("affreux",100,10);
  // ...
}
```

L'instanciation d'un objet va donc se décomposer en 3 étapes:

1. la création de l'objet dans le tas. Nous pouvons voir la l'effet du mot-clef "new". l'objet est créé, l'état des variables d'instances qui le compose sont fixées par défaut (cf figure 1).
2. Les expressions d'initialisations sont appliquées. Ici, seule la variable d'instance `exp` en définit une, ce qui a pour effet de faire passer sa valeur de l'état 0 dans l'état 0.

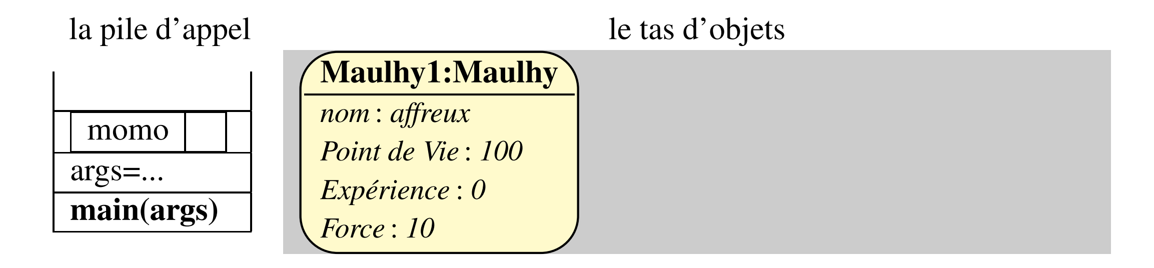


FIG. 3.22 – *new Mauhly("affreux",100,10)*

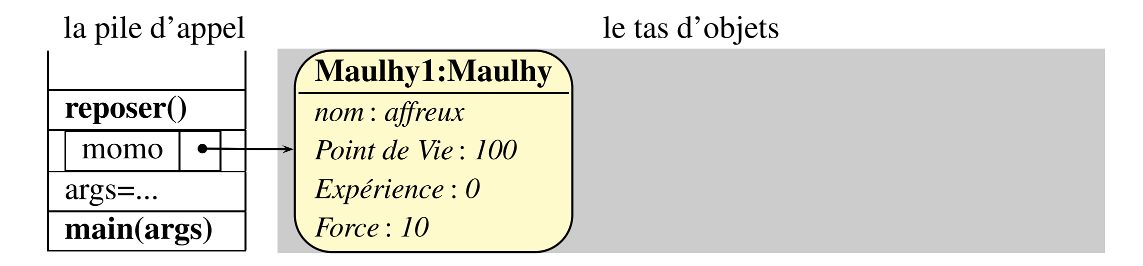


FIG. 3.23 – *Mauhly momo=new Maulhy("affreux",100,10)*

3. Le constructeur est appelé, à l'image d'une méthode d'instance sur l'objet nouvellement créé. l'état de l'objet est changé conformément au code du contructeur (cf figure 3).
4. L'objet est désormais instancié et initialisé, l'affectation de la référence `momo` peut-être réalisée (cf figure 3.33).

Une erreur classique de compilation consiste à écrire le programme figure 3.24.

En compilant, on obtient :

```
% javac *.java
Main.java:3: cannot resolve symbol
symbol  : constructor Mauhly  (java.lang.String,int,int)
location: class Mauhly
        Mauhly momo=new Mauhly("affreux",10,100);
                    ^
1 error
```

Attention la présence de `void` dans la définition de ce pseudo-constructeur le transforme en simple méthode d'instance et donc ne peut être appelée en tant que contructeur.

Si aucun constructeur n'est définit dans une classe, le compilateur en fournit un par défaut. Le constructeur par défaut pour la classe `Mauhly` est le suivant:

```
class Mauhly {
    public Mauhlhy() {
    }
}
```

```
  public static void main(String args []) {
    Mauhly momo=new Mauhly("affreux",100,10);
    // ...
  }
class Mauhly {
    private String nom;
    private int force;
    private int vie;
    private int exp=0;

    public mylabelrvoidvoid Mauhly(String nomp, int forcep, int viep) {
        nom=nomp;
        force=forcep;
        setVie(viep);
    }
    // ...
}
```

FIG. 3.24 – *Erreur classique sur les constructeurs*

Par contre, si un constructeur par défaut est définit, alors il doit être utilisé. Par exemple, si j'essaye de compiler le programme suivant:

```
class Main2 {
    public static void main(String args []) {
        Mauhly momo=new Mauhly();
        System.out.println(momo.getVie());
        momo.reposer();
        System.out.println(momo.getVie());
    }
}
```

J'obtiens l'erreur suivante:

```
% javac Main2.java
Main2.java:3: cannot resolve symbol
symbol  : constructor Mauhly  ()
location: class Mauhly
        Mauhly momo=new Mauhly();
                    ^
1 error
```

En effet, le constructeur par défaut n'existe pas puisque j'en ai définit un avec 3 paramètres.

Il est possible de définir plusieurs constructeurs pour une même classe en utilisant la surcharge paramètrique ou "overloading" en anglais. Il est possible en Java de définir deux méthodes de même nom, mais avec une liste de paramètres différentes. Le nombre de paramètres peut être différent et/ou le type des paramètres peut être différents.

```
class Mauhly {
    private String nom;
    private int force;
    private int vie;
    private int exp=0;

  // Attention. Cette méthode est bien un constructeur
  public Mauhly(String nomp, int forcep, int viep) {
       nom=nomp;
       force=forcep;
       setVie(viep);
   }

   // Attention. Cette méthode n'est pas constructeur. C'est juste une
   // méthode d'instance normale. Le constructeur et cette méthode
   // peuvent cohabiter. Il n'y pas de confit de nommage.
   public mylabelrvoidvoid Mauhly(String nomp, int forcep, int viep) {
       nom=nomp;
       force=forcep;
       setVie(viep);
   }
   // ...
   public int getVie() {
     return vie ;
   }
   private void setVie(int viep) {
     vie=viep;
   }
   public void reposer() {
     setVie(getVie()+10);
   }
}
```

FIG. 3.25 – *Constructeur et méthode d'instance*

Le nom d'une méthode avec ses paramètres typés constitue sa *signature*. Attention, le type de retour d'une méthode ne fait pas partie de la signature. Deux méthodes ne peuvent avoir la même signature dans la même classe. Dans l'exemple figure 3.26, nous utilisons la surcharge paramètrique pour la définition de plusieurs constructeurs et pour la méthode reposer.

La résolution de la surcharge paramètrique pour un appel donné se fait à la compilation. Le compilateur selectionne parmi les différentes méthodes candidates celles qui ont le même nombre de paramètres, puis choisit celle dont le type des paramètres effectifs correspond au type des paramètres formel[4].

Nous reviendrons plus tard sur les problèmes liés à la surcharge paramètrique (cf chapitre **??**).

4. Nous verrons dans le chapitre sur les conversions **??** que la résolution de la surcharge paramètrique peut-être plus complexe que ce que nous présentons ici.

```
class Mauhly {
    private String nom;
    private int force;
    private int vie;
    private int exp=0;

    public Mauhly(String nomp,
                  int forcep,
                  int viep) {
        nom=nomp;
        force=forcep;
        vie=viep;
    }

   public Mauhly(String nomp) {
        nom=nomp;
    }
   public Mauhly(int forcep) {
        force=forcep;
    }

    public Mauhly(String nomp, int forcep) {
        nom=nomp;
        force=forcep;
    }

    // ...
    private void setVie(int viep) {
      vie=viep;
    }
    public void reposer() {
      setVie(vie+10);
    }

    // se reposer plusieurs fois ...
    public void reposer(int nbtour) {
        for (int i=0;i<nbtour;i++) {
            reposer();
        }
    }
}
```

FIG. 3.26 – *Surcharge paramétrique sur les constructeurs*

3.5 Le mot clef this

Nous avons déjà vu “this” apparaitre.

3.6 Références

Une référénce est lien typé sur un objet typé. Le type de la référence aussi appelé type statique doit être compatible avec le type de l’objet référencé aussi appelé type dynamique. Une référence peut référencer $null$ ou un objet dont le type dynamique est compatible avec le type statique de la référence.

Avant d’aller plus nous allons complexifier quelque peu notre exemple. Pour l’instant notre monstre “mauhly” n’est caractérisé que par son nom, sa force, sa vie et son expérience. Rappelons nous qu’il peut aussi porter une arme, une armure et un bouclier.

3.6.1 Références et graphes d’objets

Dans un monde objet, Arme, Armure et bouclier sont des objets. Si un “mauhly” porte une arme alors l’objet “Mauhly” référence un objet arme.

Une armure et un bouclier sont caractérisés par un *coefficient de protection* $prot$, une arme par un *coefficient de destruction* $destr$. Ces coefficients sont constants : ils ne peuvent pas varier au cours du jeu.

```
class Main {
    public static void main(String args []) {
        Mauhly a=new Mauhly("affreux",15,25);
        Mauhly pb=new Mauhly("pas beau");
        Mauhly apb=new Mauhly("affreux pas beau",10);
        a.reposer();
        a.reposer(3);
    }
}
```

```
class Mauhly {
    private String nom;
    private int force;
    private int vie;
    private int exp=0;

    public Mauhly(String nomp, int forcep, int viep) {
        nom=nomp;
        force=forcep;
        vie=viep;
    }
    public Mauhly(String nomp) {
        nom=nomp;
    }
    public Mauhly(int forcep) {
        force=forcep;
    }
    public Mauhly(String nomp, int forcep) {
        nom=nomp;
        force=forcep;
    }
    // ...
    private void setVie(int viep) {
      vie=viep;
    }
    public void reposer() {
      setVie(vie}10);
    }
    // se reposer plusieurs fois ...
    public void reposer(int nbtour) {
        for (int i=0;i<nbtour;i}}) {
            reposer();
        }
    }
}
```

Nous introduisons la classe $Arme$ et $Armure$ dans le jeu.

```
class Main {
    public static void main(String args []) {
        Mauhly m=new Mauhly("affreux",15,25);
        m.setArme(new Arme(5));
        m.setArmure(new Armure(6));
    }
}
```

La figure 3.30 montre le diagramme de classe de notre application. Il est assez facile de faire correspondre le code ci-dessus avec ce diagramme. Il faut bien remarquer que les variables d'instance de la classe "mauhly" référençant les classes armes et armures sont représentées par des relations.

```
class Mauhly {
    private String nom;
    private int force;
    private int vie;
    private int exp=0;

    public Mauhly(String nom, int force,
                  int vie) {
        nom=this.nom;
        force=this.force;
        vie=this.vie;
    }

    public Maulhy(String nom) {
        this(nom,0,0);
    }

    public Mauhly(String nom, int force) {
        this(nom,force,0);
    }

    public int getForce() {
        return this.force;
    }
    public int getExp() {
        return this.exp;
    }
    public int getVie() {
      return this.vie ;
    }
    private void setVie(int vie) {
      this.vie=vie;
    }
    public void reposer() {
      this.setVie(this.getVie()+10);
    }
}
```

FIG. 3.27 – *this*

```
class Arme {
    private int destr=0;
    public Arme(int destr) {
        this.setDestr(destr);
    }

    public int getDestr() {
        return destr;
    }

    private void setDestr(int destr) {
        this.destr=destr;
    }
}
```

```
class Armure {
    private int prot=0;
    public Armure(int prot) {
        this.setProt(prot);
    }

    public int getProt() {
        return prot;
    }

    private void setProt(int prot) {
        this.prot=prot;
    }
}
```

FIG. 3.28 – *Classes Arme et Armure*

La figure 3.31 montre le diagramme de séquence de notre application.

La figure 3.32 montre le diagramme de séquence de notre application.

Ces différents diagramme permettent d'avoir une vue globale de l'application. Le diagramme de classe illustre la statique du programme, les diagrammes de séquences et de collaboration la dynamique.

3.6.2 Affectation de références

Le programme de la figure 3.34 illustre les problèmes liés à l'affectation des références.

```
class Mauhly {
    private String nom;
    private int force;
    private int vie;
    private int exp=10;

    private Arme arme=null;
    private Armure armure=null;

    public Mauhly(String nomp,
                  int forcep,
                  int viep) {
        nom=nomp;
        force=forcep;
        vie=viep;
    }

    public void setArme(Arme arme) {
        this.arme=arme;
    }

    public Arme getArme() {
        return arme;
    }

    public void setArmure(Armure armure) {
        this.armure=armure;
    }

    public Armure getArmure() {
        return armure;
    }
    // ...
}
```

FIG. 3.29 – *Construire des réseaux d'objets*

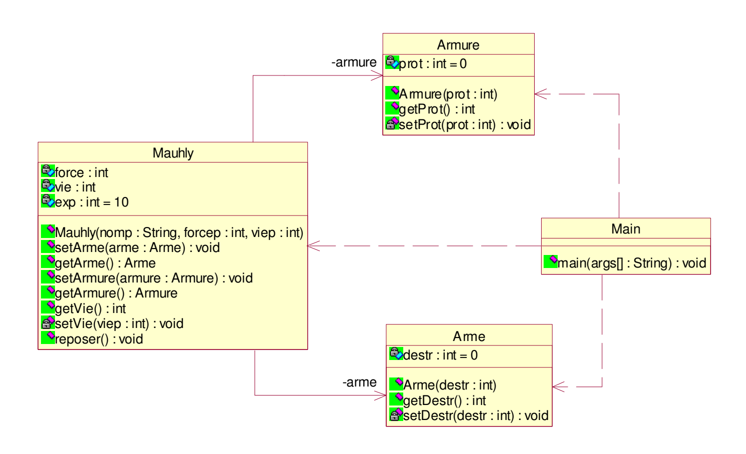


FIG. 3.30 – *Diagramme de classe*

La figure 3.35 illustre l'état de la machine juste avant d'exécuter $m1 = m2$.

La figure 3.36 illustre l'état de la machine juste après l'exécution de $m1 = m2$.

Il faut bien comprendre ici qu'affecter $m2$ à $m1$ ne recopie l'état de l'objet $m2$ dans $m1$, seule la référence $m1$ qui référençait l'objet $m1$ référence desormais l'objet référencé par $m2$.

L'incompréhension de l'affection des rérérences est source d'erreur bien connue chez les débutants.

3.6.3 Passage de références en paramètres

Cette particularité d'affection des références se retrouve de la même manière lors du passage des paramètres. Dans la figure 3.37, la méthode *combattre* prend

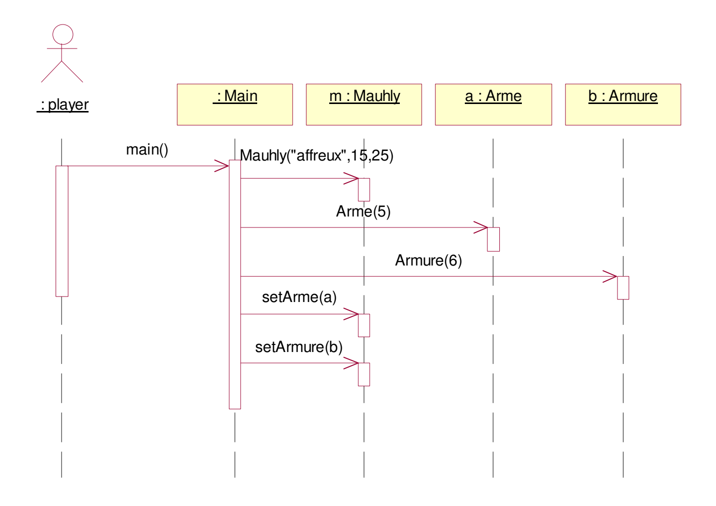


FIG. 3.31 – *Diagramme de séquence*

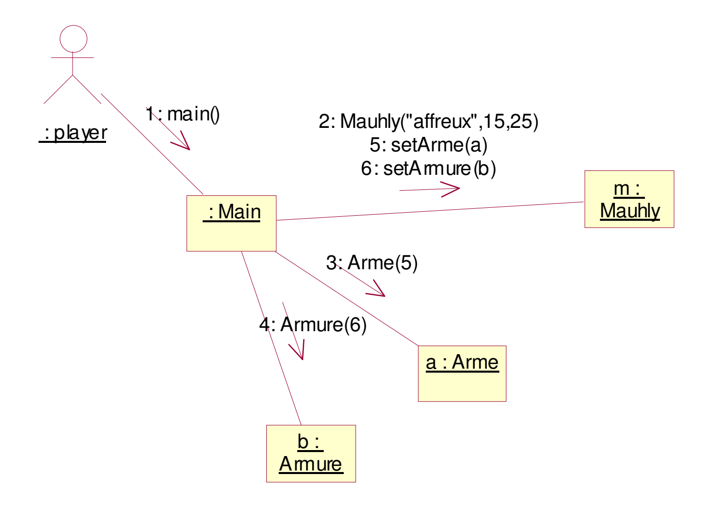


FIG. 3.32 – *Diagramme de collaboration*

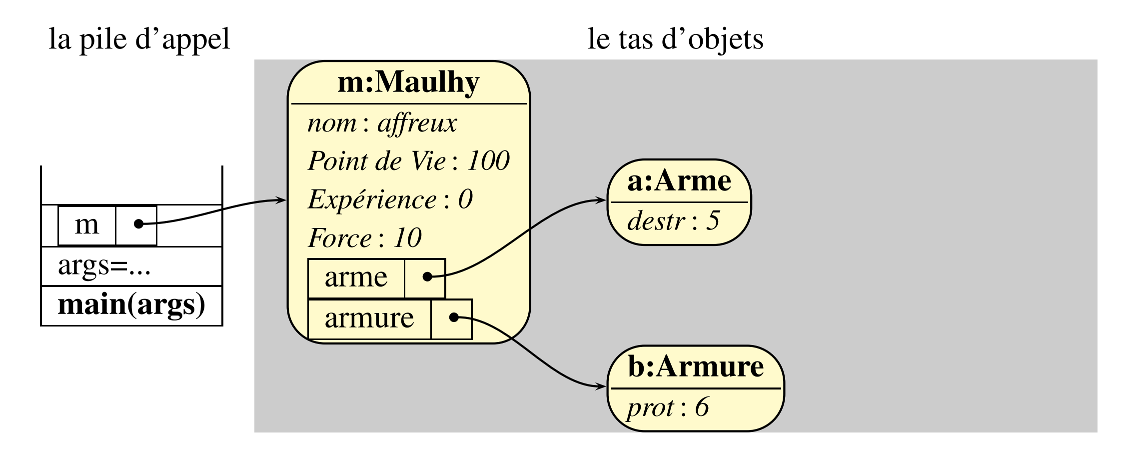


FIG. 3.33 – *Etat de la machine vituelle avant la fin de main()*

```
class Main {
    public static void main(String args []) {
        Mauhly m1=new Mauhly("affreux",15,25);
        Mauhly m2=new Mauhly("pas␣bo",10,30);
        Mauhly m3=m2;
        m1=m2;
    }
}
```

FIG. 3.34 – *Affectation de références*

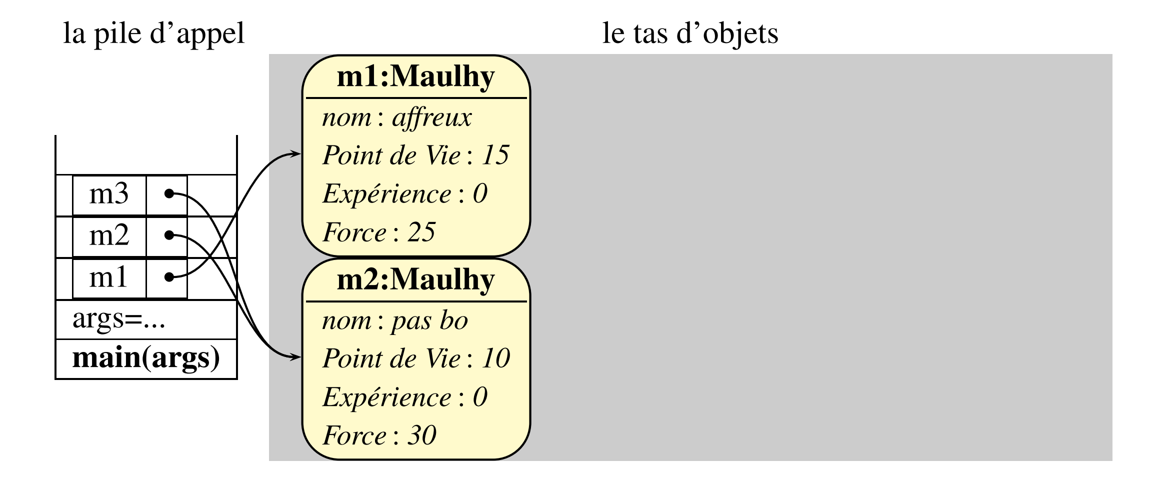


FIG. 3.35 – *Affectation de références*

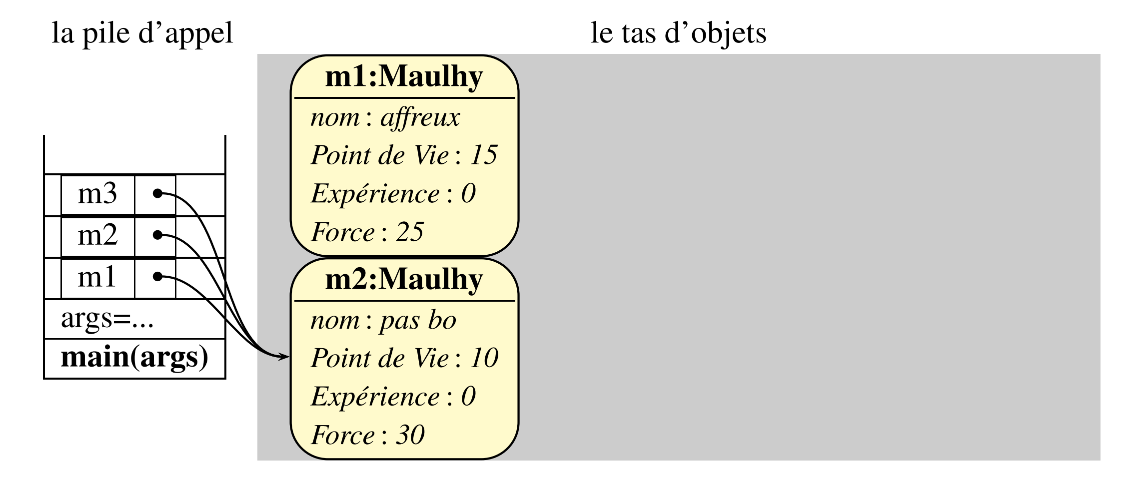


FIG. 3.36 – *Affectation de références*

```
class Mauhly {
    private String nom;
    private int force;
    private int vie;
    private int exp=10;

    public Mauhly(String nomp, int forcep, int viep) {
        nom=nomp;
        force=forcep;
        vie=viep;
    }

    public void combattre(Mauhly ennemi) {
        ennemi.setVie(ennemi.getVie()−this.force);
    }

    public int getVie() {
      return vie ;
    }
    private void setVie(int viep) {
      vie=viep;
    }
    public void reposer() {
      setVie(getVie()+10);
    }
}
```

FIG. 3.37 – *Passage de référence en paramètres*

une référence sur un objet de classe “Mauhly” en paramètre. Lorsque cette méthode est appelée, le passage de paramètre s’effectue par référence, i.e. l’objet passé en paramètre n’est pas copié, $ennemi$ n’est rien de plus qu’un référence supplémentaire sur l’objet désigné à l’appel.

Le programme principal suivant permet de visualiser le passage d’un objet en paramètre d’un appel de méthode.

```
class Main {
    public static void main(String args[]) {
        Mauhly m1=new Mauhly("affreux",15,25);
        Mauhly m2=new Mauhly("pas␣bo",10,30);
        m1.combattre(m2);
        System.out.println(m2.getVie());
    }
}
```

Avant d’exécuter l’appel de méthode $m1.combattre(m2)$, l’état de la machine virtuelle est celui représenté dans la figure 3.38.

La figure 3.39 illustre l’état de la machine pendant l’exécution de la méthode $combattre$

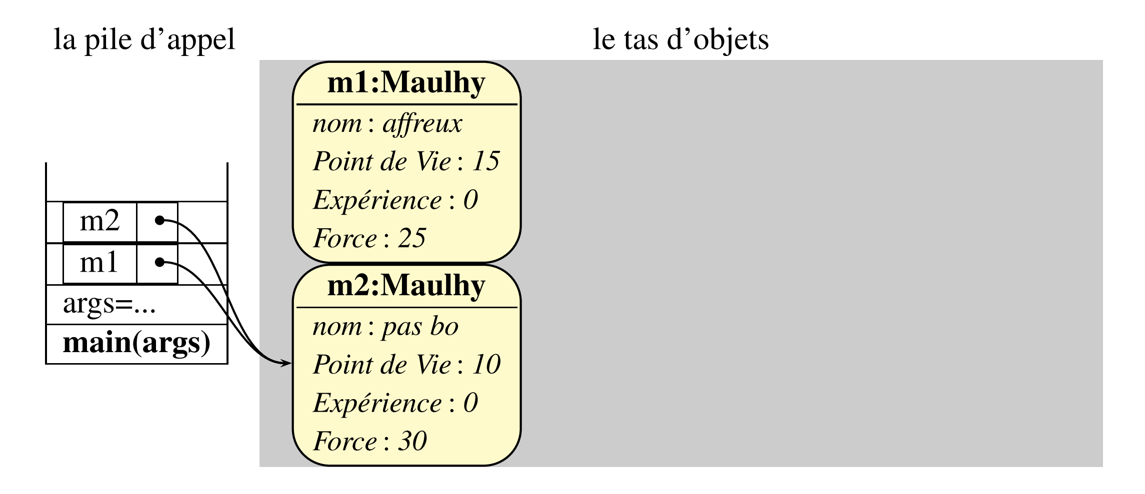


FIG. 3.38 – *Passage de référence en paramètres*

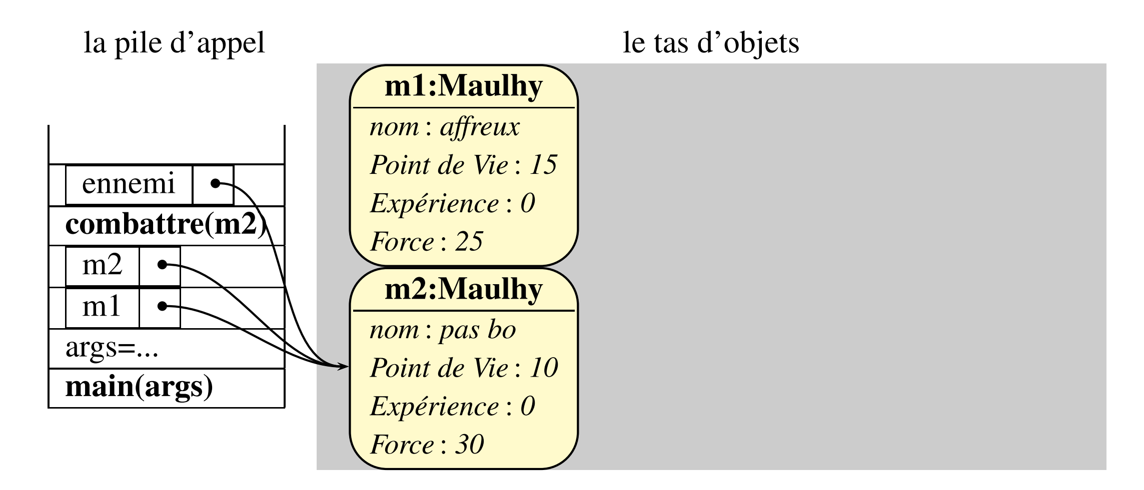


FIG. 3.39 – *Passage de référence en paramètres*

Il faut bien comprendre que tout passage d'objets en paramètre se fait par référence[5]auf lorsqu'on utilise les invocations de méthodes à distance (RMI) dans un cas bien particulier. Tout changement d'état effectué dans le corps de la méthode sera visible par la méthode appelante. Ce qui est tout à fait normal puisque les deux méthodes référence et donc se partage le même objet.

Dans notre exemple, l'exécution de la méthode $combattre$ a pour effet de diminuer les points de vie du monstre combattu. à la fin de l'exécution de la méthode combattre, $m1.getVie()$ imprime $30 - 10 = 20$.

Ce comportement n'est pas toujours souhaitable, il peut être utile parfois que la méthode appelée n'ait aucun effet de bord i.e. que tout changement opéré par elle ne soit visible que par elle. Dans ce cas, il faut faire en sorte qu'elle n'opère que sur des copies d'objets et non sur les objets originaux.

5. s

```
class Mauhly {
 private String nom;
 private int force;
 private int vie;
 private int exp=10;

 private Arme arme=null;
 private Armure armure=null;

 public Mauhly(String nomp,
               int forcep,
               int viep) {
      nom=nomp;
      force=forcep;
      vie=viep;
 }

 // un copy-constructeur dont le
 // code n'est pas un exemple a suivre ...
 // Mais c'est pour la bonne cause !
    public Mauhly(Mauhly m) {
         nom=m.nom;
         force=m.force;
         vie=m.vie;
         exp=m.exp;
         arme=m.arme;
         armure=m.armure;
    }
    // ...
}

class Main {
    public static void main(String args[]) {
         Mauhly m1=new Mauhly("affreux",15,25);
         m1.setArme(new Arme(5));
         m1.setArmure(new Armure(6));
         Mauhly m2=new Mauhly(m1);
    }
}
```

FIG. 3.40 – *Copie d'objet*

3.6.4 Référence et copies d'objets

Copier un objet n'est pas chose facile dans tout langage objet. En effet, comme nous l'avons vu, un objet n'est souvent qu'un composant d'un réseau d'objet. Dans ce cas faut-il copier seulement l'objet lui-même ? Ou le sous-graphe d'objet atteignable à partir de cet objet[6] ? La réponse à cette question est dépendante de l'application que nous voulons écrire.

En java, si le programmeur veut à un moment ou autre dupliquer un objet, c'est à lui d'écrire le code permettant de dupliquer cet objet. Il y a plusieurs méthodes pour faire cela. La méthode privilégiée par Java est la redéfinition de la méthode $clone$. Redéfinir signifie ici qu'elle existe déjà quelque part comme nous le verrons dans le chapitre sur l'héritage, mais que le programmeur choisit de la spécialiser pour les besoins propres des objets de cette classe. L'utilisation de cette méthode demande plus de connaissance sur le langage que nous n'en disposons pour l'instant. Par contre, nous pouvons tout à fait utiliser une technique bien connue des programmeur C++ les copy-constructeurs. Cette technique n'est pas incompatible avec la gestion des clones privilégiée en Java. Il s'agit d'utiliser la surcharge paramétrique (cf ??) pour définir un nouveau constructeur prenant en paramètre un objet de la classe (cf figure **??**).

La figure 3.41 illustre l'action du copy-constructeur définit dans la class $Mauhly$. Ce copy-constructeur recopie variable d'instance pour variable d'instance l'état de l'objet original. Mais bien évidemment, dans ce programme il n'existe qu'une seule

6. on parle alors de copie-profonde ou deep-copy

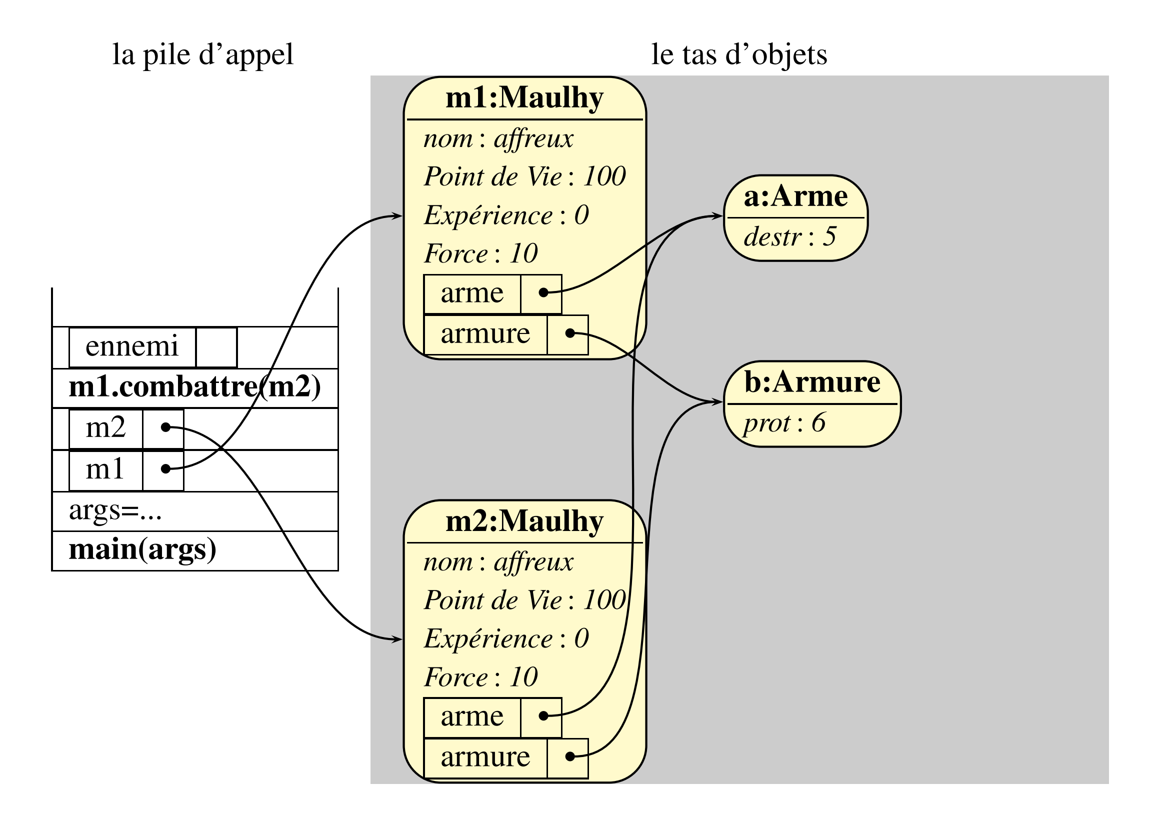


FIG. 3.41 – *Graphe d'objet après copie partielle*

arme et armure partagés par les deux instances de $Mauhly$. La copie ici n'est pas profonde. Si nous voulons obtenir une copie complète, il faut dupliquer les objets arme et armure. La figure 3.43 montre comment nous avons définit les copy-constructeur dans les classes arme et armure et comment nous avons modifié le copy-constructeur de la $Mauhly$ en conséquence.

La figure 3.43 montre l'état du graphe d'objets après exécution du copy-constructeur définit dans $Mauhly$. Il faut bien remarquer cette comment l'objet $m1$ a été dupliqué totalement et non partiellement (cf 3.41

3.6.5 Egalité des références

Ces problèmes d'affectation de références, de copies entraine dans leurs sillages les problèmes d'égalité. En effet si j'ai deux références a et b sur des objets de même type. Que signifie exactement $a == b$[7] ?

Est-ce l'égalité des références ou des objets référencés ? Le programme de la figure 3.44 répond à la question.

En java, l'égalité entre deux références porte sur la référence elle-même. Si les deux références a,b référencent le même objet, alors $a == b$ est vrai, sinon $a == b$

7. A ne pas confondre avec $a = b$ l'affectation

```
class Mauhly {
    private String nom;
    private int force;
    private int vie;
    private int exp=10;

    private Arme arme=null;
    private Armure armure=null;

    // ...

    public Mauhly(Mauhly m) {
        nom=m.nom;
        force=m.force;
        vie=m.vie;
        exp=m.exp;
        arme=new Arme(m.arme);
        armure=new Armure(m.armure);
    }
    // ....
}

class Arme {
    private int destr=0;
    public Arme(int destr) {
        this.setDestr(destr);
    }

    // copy constructeur
    // (mieux écrit du point de
    // vue de l'encapsulation)
    public Arme(Arme a) {
        this.setDestr(a.getDestr());
    }

    // ...
}
```

FIG. 3.42 – *Copy-constructeur: copie profonde*

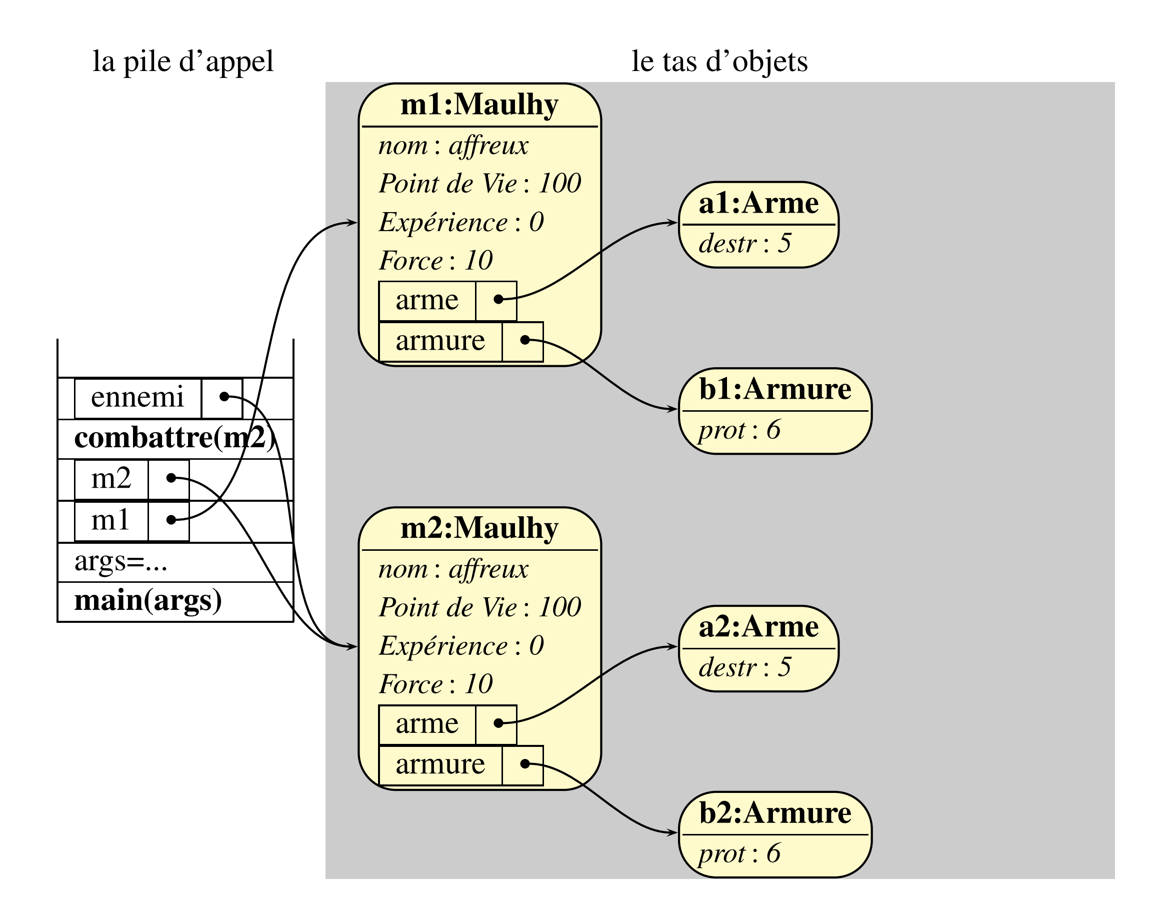


FIG. 3.43 – *graphe d'objets après copies profondes*

```
class Main {
  public static void main(String args []) {
      Mauhly m1=new Mauhly("affreux",15,25);
      Mauhly m2=new Mauhly("affreux",15,25);
      System.out. println ((m1==m2)); // false
      m1=m2;
      System.out. println ((m1==m2)); // true
  }
}
```

FIG. 3.44 – *Egalité des références*

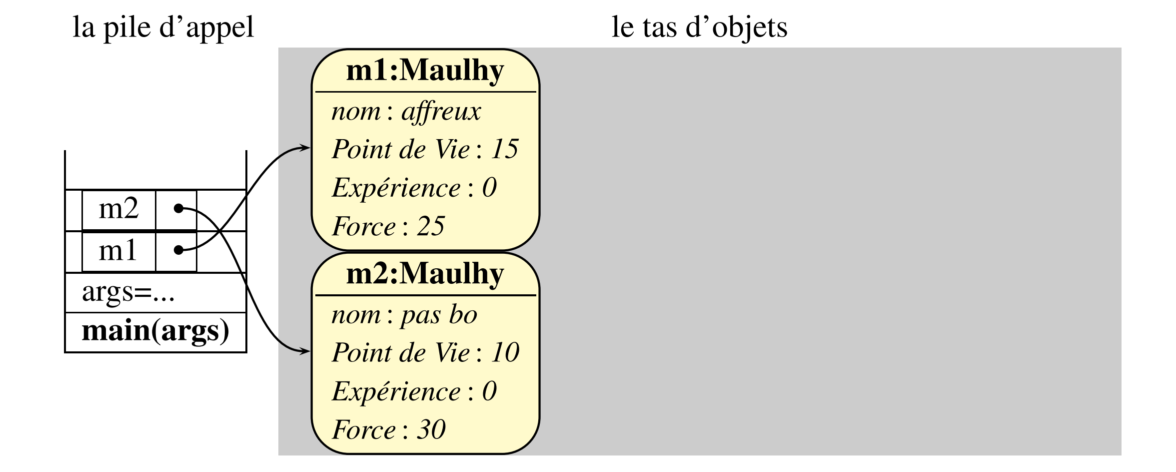


FIG. 3.45 – $m1 == m2$ *est faux*

est faux même si l'état des objets référencé est strictement identique comme c'est le cas ici.

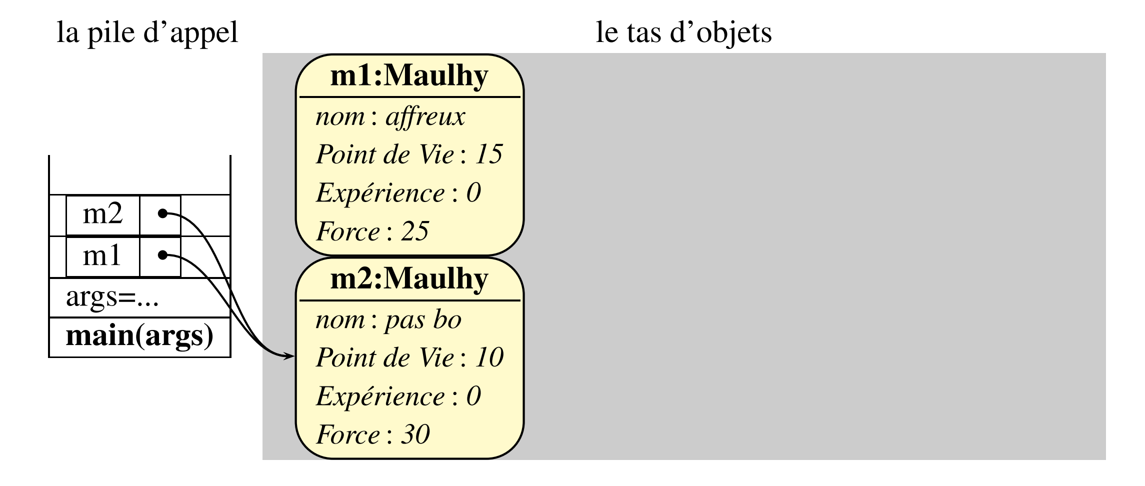


FIG. 3.46 – $m1 == m2$ *est vrai*

Si cette manière de déterminer si deux références sont égales est compréhensible, il est tout aussi utile de déterminer si deux objets sont identiques. $a == b$ implique $a.equals(b)$ par contre l'inverse est faux.

Dans ce cas, nous retombons sur les mêmes problèmes que ceux rencontrés pour la copie d'objets. L'égalité entre deux objets est dépendante de l'application que nous voulons écrire. Par exemple, deux monstres de classe $Mauhly$ peuvent être identiques si ils ont juste le même nom ou si tous les objets qui composent le sous-graphes atteignable depuis ces objets sont identiques.

Comme la méthode $clone$ pour les copies, il existe une méthode $equals$ pour determiner l'egalité entre objets. Cette relation doit être:

reflexive $x.equals(x) \rightarrow true$,

symétrique si $x.equals(y) \rightarrow true$ alors $y.equals(x) \rightarrow true$,

transitive si $x.equals(y) \rightarrow true$ et $y.equal(z) \rightarrow true$ alors $x.equals(z) \rightarrow true$
$x.equals(null) \rightarrow false$.

Si nous voulons gérer l'egalité entre deux objets monstres, nous devons redéfinir la méthode equals pour deux monstres de classe $Mauhly$.

Une façon naturelle de faire serait de définir une méthode $booleanequals(Mauhlym)$. Nous verrons dans le chapitre sur la liaison dynamique que cette définition de la méthode $equals$ pose en fait un problème fondamental de sureté des types liés aux langages objets à base de classes. Pour l'instant, il nous semble naturel de faire de façon, nous choisissons de continuer dans cette voie. La figure **??** illustre la définition de notre méthode $equal$

La méthode $equals$ définie dans $Mauhly$ fait une comparaison variable d'instance par variable d'instance des états respectifs des objets. La comparaison des variables d'instances de type primitifs $force$,vie,exp ne pose pas de problèle. Par contre, la variable d'instance nom est en fait une référence sur objet de classe $String$. Si le compilateur est assez malin pour ne créer qu'un seul objet "String" pour les déclarations Mauhly m1=**new** Mauhly("affreux",15,25) et Mauhly m2=**new** Mauhly("affreux",15,25);, en revanche pour $m3$, nous forçons la création d'un nouvel objet $String$.

La figure **??** montre l'état du graphe d'objet après la création de $m1$,$m2$,$m3$.

La méthodes $equals$ fait son teste l'égalité des références vers des objets $String$ et non l'égalité des objets $string$ eux-même.

Pour que ce même programme imprime "true, true", il faut réécrire la méthode $equals$ de manière plus adéquate (cf figure **??**).

Bien comprendre la manipulation des références est indispensable. Une connaissance partielle des règles régissant l'affectation, le passage en paramètre, la copie et l'égalité des références est une source inépuisable de malfonctionnement en tous genres.

3.7 Encapsulation

Accés aux valeurs private. méthode combattre sur un monstre prenant en paramètre un autre monstre.

```
class Mauhly {
    private String nom;
    private int force;
    private int vie;
    private int exp=10;

    public Mauhly(String nomp, int forcep, int viep) {
        nom=nomp;
        force=forcep;
        vie=viep;
    }
    public boolean equals(Mauhly m) {
        return ((nom==m.nom) && (force==m.force) &&
                (vie==m.vie) && (exp==m.exp));
    }
    // ...
}
class Main {
    public static void main(String args[]) {
        Mauhly m1=new Mauhly("affreux",15,25);
        Mauhly m2=new Mauhly("affreux",15,25);
        Mauhly m3=new Mauhly(new String("affreux"),15,25);
        System.out.println((m1.equals(m2))); // true
        System.out.println((m1.equals(m3))); // false
    }
}
```

FIG. 3.47 – *m1.equals(m2)*

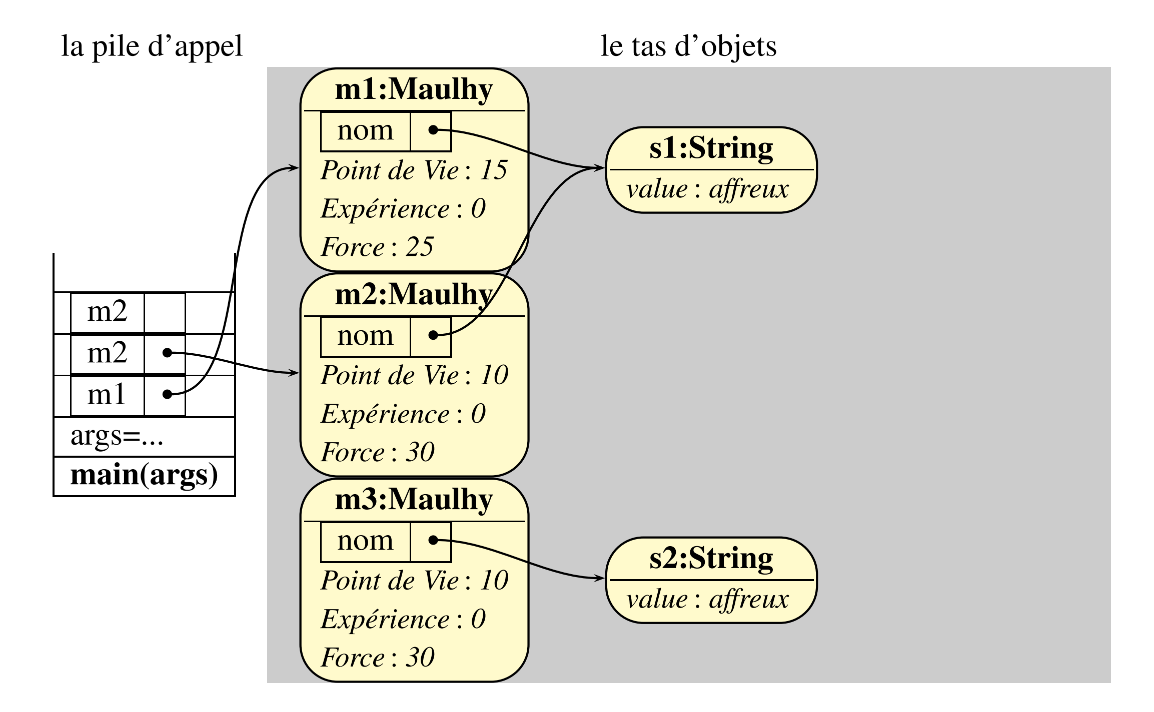


FIG. 3.48 – *Equal ou presque...*

```
class Mauhly {
    private String nom;
    private int force;
    private int vie;
    private int exp=10;

    public Mauhly(String nomp, int forcep, int viep) {
        nom=nomp;
        force=forcep;
        vie=viep;
    }

    // equals pour les références !
    // == pour les type primitifs .
    public boolean equals(Mauhly m) {
        return ((nom.equals(m.nom)) && (force==m.force) &&
                (vie==m.vie) && (exp==m.exp));
    }
    // ...
}
```

FIG. 3.49 – *m1.equals(m2)*

```
class Mauhly {
    private String nom;
    private int force;
    private int vie;
    private int exp=10;

    public Mauhly(String nomp, int forcep, int viep) {
        nom=nomp;
        force=forcep;
        vie=viep;
    }

    public void combattre(Mauhly ennemi) {
        // La bonne façon de faire
        ennemi.setVie(ennemi.getVie()-this.force);
        // Le mauvais coté d'une encapsulation
        // basée sur les classes ...
        ennemi.vie=ennemi.vie-this.force;
    }
    // ...
}
```

FIG. 3.50 – *Encapsulation basée sur les classes*

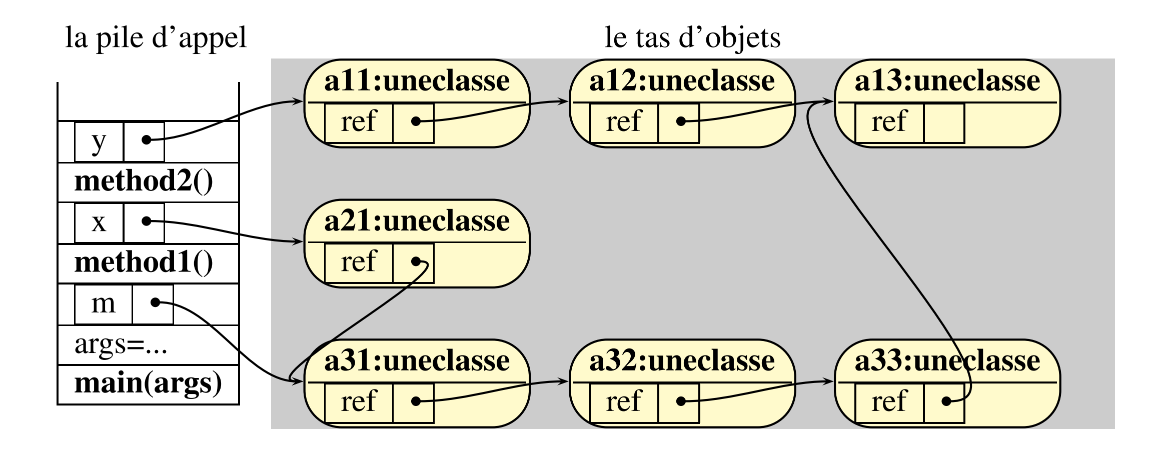


FIG. 3.51 – *Etat de la machine virtuelle à un instant donné*

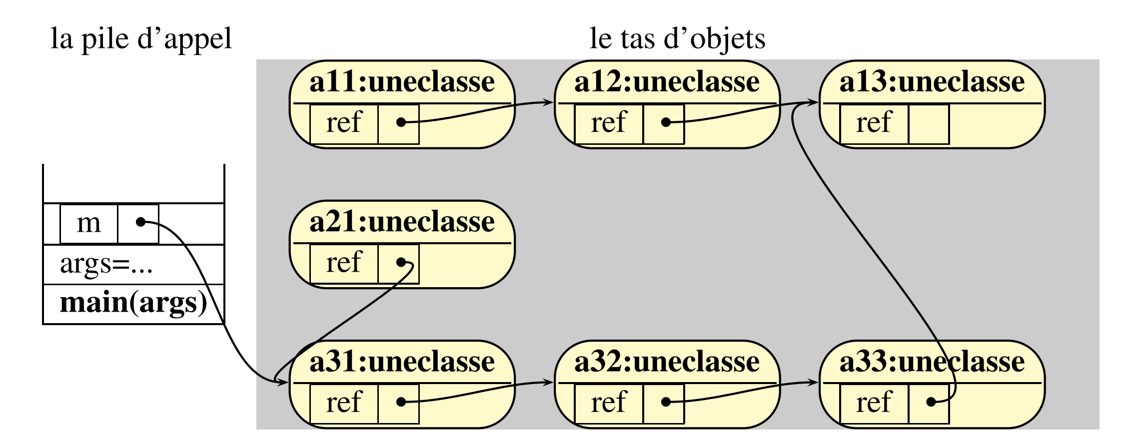


FIG. 3.52 – *Etat de la machine virtuelle à un instant donné*

faire apparaitre les problèmes d'encapsulation basée sur les classes en java ...

3.8 Mort des objets : Ramasse-miettes

ramasse-miettes. Jolis diagrammes pour illustrer le mark and sweep...

Objet vivants et morts.

La figure 3.51 montre un état possible de la machine virtuelle.

la figure 3.52 montre un état ultérieur où les appels de méthodes “method2” et “method1” sont terminés. Supposons que le ramasse-miette se déclanche à ce moment précis.

Le ramasse-miette se comporte de la façon suivante. Il parcourt la pile d'appel pour trouver les racines du graphe du graphe d'objet. Ici en l'occurence la racine en question est la référence m. Partant de cette référence, il parcourt le graphe d'objet et marque tous les objets atteignables.

Les objets non atteignables ne peuvent plus faire l'objet d'appel de méthodes, il

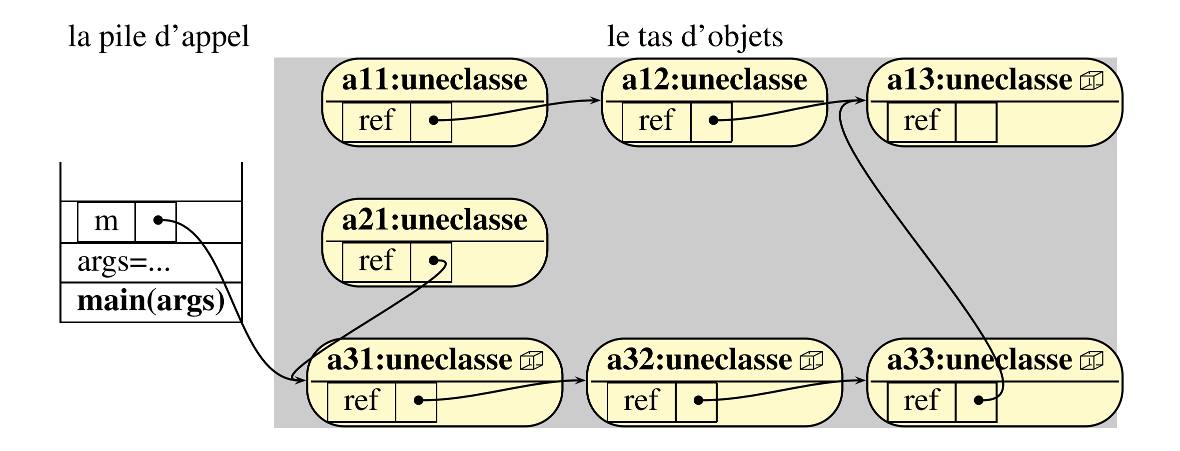


FIG. 3.53 – *Etat de la machine virtuelle à un instant donné*

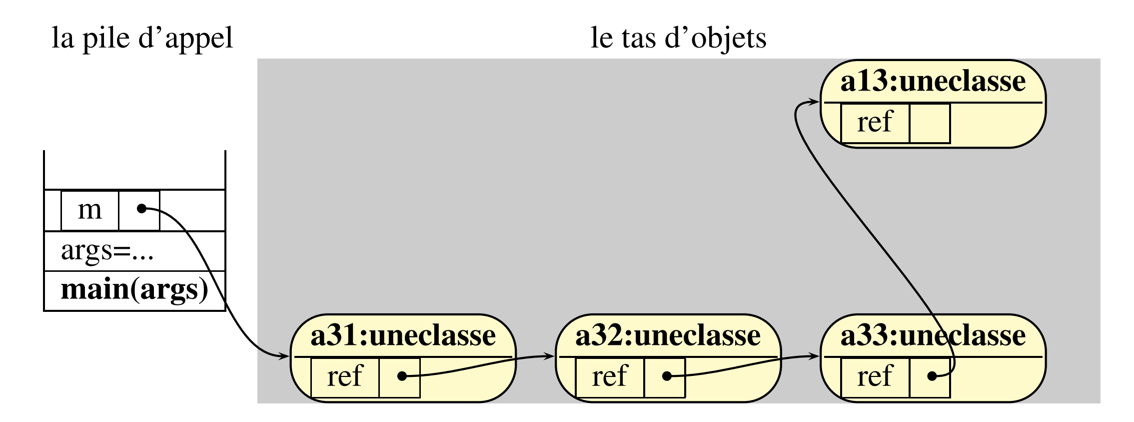


FIG. 3.54 – *Etat de la machine virtuelle à un instant donné*

sont donc ramassés par le ramasse-miette. Ils sont détruits (cf figure 3.53).

Avant leur destruction, la ramasse-miette donne une dernière chance à l'objet devant être détruit en appelant la méthode $finalize()$, si elle existe.

3.9 Constantes, variables et méthodes de classes

Pour l'instant nous avons vu qu'une méthode ou une variable étaient toujours associées à un objet. Nous les avons d'ailleurs appelées variables et méthodes d'instances. Il est possible d'associer des méthodes et des variables aux classes directement. Elles se comportent alors presque comme des variables globales et des fonctions sauf qu'elles sont déclarées dans l'espace de nommage d'une classe. Les mécanismes d'encapsulation jouent tout de même leur rôle.

Le nom de la classe peut-être utilisé comme receveur.

retour sur `main` et `this`, les constantes

```
class Mauhly {
    private String nom;
    private int force;
    private int vie;
    private int exp=10;

    private Arme arme=null;
    private Armure armure=null;

    public Mauhly(String nomp, int forcep,
                  int viep) {
        nom=nomp;
        force=forcep;
        vie=viep;
    }
    // ...
    public void finalize () {
        System.out.println(this.nom}"␣ramassé");
    }
    // ...
}
class Main {
  public static void main(String args[]) {
    Mauhly m1=new Mauhly("affreux",15,25);
    m1.setArme(new Arme(5));
    m1.setArmure(new Armure(6));
     while (true) {
       Mauhly m2=new Mauhly("pa␣bo",18,30);
       m2.setArme(new Arme(10));
       m2.setArmure(new Armure(10));
     }
  }
}
```

FIG. 3.55 – *Activer le ramasse-miettes*

```
class Hero {
    private String nom="Hero";
    private int force;
    private int vie;
    private int exp=10;

    private static Hero hero=null;

    private Hero() {
        Random r=new Random();
        force=r.nextInt(10);
        vie=r.nextInt(10);
    }

    public static Hero getInstance() {
        if (hero==null) {
            hero=new Hero();
        }
        return hero
    }
    public void combattre(Mauhly ennemi) {
        ennemi.setVie(ennemi.getVie()-this.force);
    }

    public int getVie() {
      return vie ;
    }
    private void setVie(int viep) {
      vie=viep;
    }
    public void reposer() {
      setVie(getVie()+10);
    }
}
class Main {
    public static void main(String args[]) {
        Hero.getInstance().reposer();
        Hero h=Hero.getInstance();
        h.reposer();
    }}
```

FIG. 3.56 – *Variables et méthodes : le singleton*

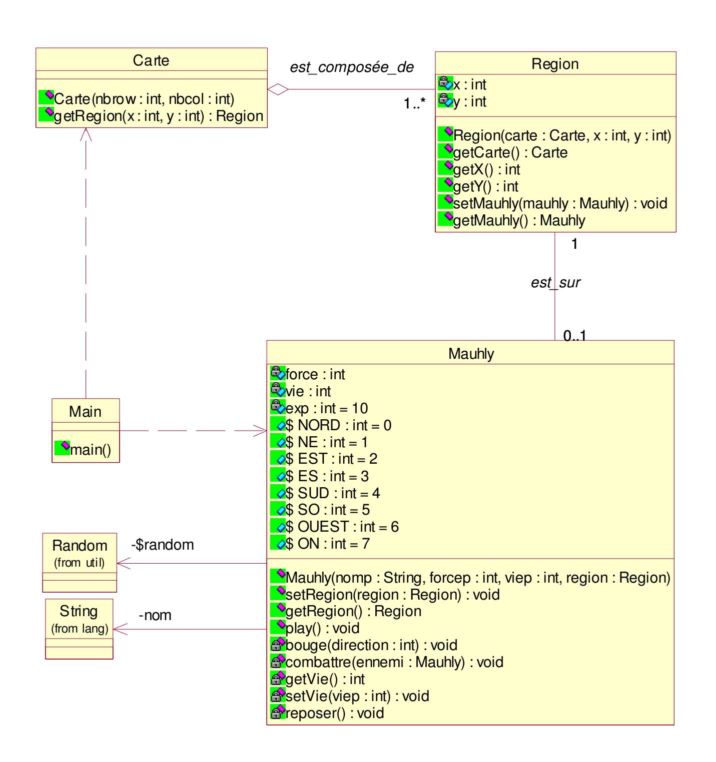


FIG. 3.57 – *Diagramme de classe pour la gestion de la carte*

schémas UML correspondants (représentation des truc statiques...)

3.10 Relations 0-n

Jusqu'à présent nous avons créés des graphes d'objets ou il n'existe que des relations 1-1 entre deux objets. Un monstre a une arme , une arme appartient ou n'appartient pas à une monstre.

Comment représenter des relations 0-n. Par exemple, si je veux représenter la carte des tortues Java. Il va forcément exister un objet carte. Une carte est-elle même composée de cases, plus précisement d'une matrice de cases. Les cases vont être elles-aussi des objets, et l'objet carte va contenir 0-n cases.

La figure 3.57 montre le diagramme de classe que nous voulons réaliser pour les tortues Java.

Il existe dans les langages objets plusieurs manière de représenter ces relations. Suivant le type de la relation que l'on a représenter; par exemple est-ce un ensemble ? une séquence ? On va s'orienter soit sur des tableaux, des vecteurs, des tables de hashage.

Toutes ces structures permettant de représenter des relations 0-n sont généralement regroupées dans un ensemble de classes appelées "collections" car elles permettent de créer des collections d'objets.

Déclaration d'un référence sur un objet tableau à deux dimensions

Instanciation d'un object tableau contenant des références non initialisées

Accès à la dimension du premier tableau. En fait, accès à une variable d'instance de l'objet tableau

Accès à la deuxième dimension. On voit déjà ici qu'un tableau à deux dimensions est un tableau de tableaux.

Affection des références contenus dans les objets tableaux sur des objets regions

Acces à un élément du tableau. le bloc try/catch est ici pour intercepter une eventuelle exception. Celle-ci peut se produire si on tente d'accéder à une cellule du tableau qui n'existe pas.

```
class Carte {
    private Region regions[][];

    public Carte(int nbrow, int nbcol) {
        regions=new Region[nbrow][nbcol];
        for (int i=0;i<regions.length;i++) {
            for (int j=0;j<regions[i].length;j++) {
                regions[i][j]=new Region(this,i,j);
            }
        }
    }
    public Region getRegion(int x, int y) {
        try {
            return regions[x][y];
        } catch (ArrayIndexOutOfBoundsException e) {
            return null;
        }
    }
}
```

FIG. 3.58 – *Déclaration; initialisation et accès dans un tableau à deux dimensions*

Ces classes collections regroupent le plus souvent ce qui fait les beaux jours des cours d'algorithmique de toute formation en informatique; structures de piles, de files, de listes chaines, de queues, avec leurs méthodes d'insertions et de tris .... Dans les langages objets toutes ces structures existent déjà dans les bibliothèques de collections.

Les classes collections sont le plus souvent une construction basée sur une structure de données primaire : les tableaux. La figure 3.58 montre comment nous réalisons la relation 0-n $Carte \rightarrow Region$ avec un tableau à 2 dimensions. En java, les tableau sont des objets. Pour manipuler un objet tableau, il nous une référence typée dessus. C'est l'objectif de la déclaration `Region regions [][]`. $regions$ est une référence sur objet tableau à deux dimensions ([][]) devant contenir des références sur des objets $Region$.

Supposons que nous instancions un objet $Carte$ en appelant le constructeur de $Carte$, `new Carte(3,3)`, l'état de la machine virtuelle est celui représenté dans la figure 3.59. La référence sur l'objet tableau est juste non-initialisée.

Il nous faut maintenant instancier l'objet tableau lui-même; c'est l'objectif de `new Region[nbrow][nbcol]`. La figure 3.60 représente l'état de la machine virtuelle après

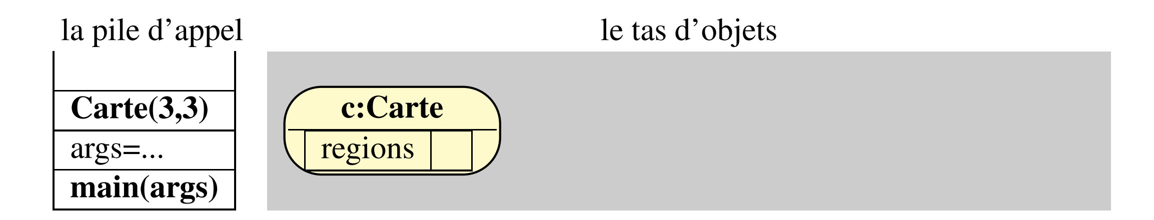


FIG. 3.59 – *Vue mémoire d'un objet tableau à deux dimensions*

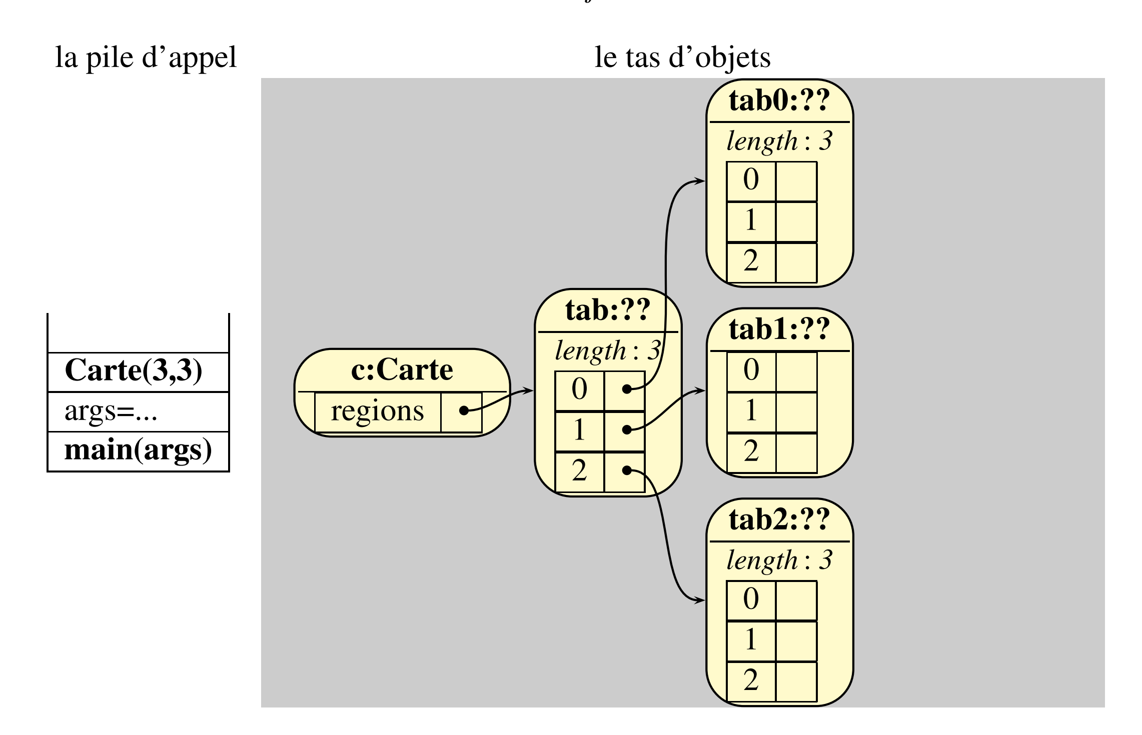


FIG. 3.60 – *Vue mémoire d'un objet tableau à deux dimensions*

la création des objets tableaux. Nous pouvons clairement nous rendre compte qu'un tableau à deux dimension est en fait un tableau de tableaux. La taille du tableau est disponible sous forme d'une variable d'instance *length* : *3* de l'objet tableau. Il est important de noter qu'un tableau ne peut contenir que des références ou des types primitifs.

Le constructeur de l'objet carte montre comment nous parcourons la structure de tableau et créons pour chaque référence un objet $Region$ correspondant (cf figure 3.61. Il faut remarquer comment nous accèdons aux tailles des objets tableaux `regions.length` et `regions[i].length`.

La méthode d'accès `public Region getRegion(int x, int y)` utilise une construction que nous n'avons pas encore rencontrée; les blocs $try - catch$. Nous reviendrons en profondeur sur ces constructions dans le chapitre 8. Pour l'instant, il nous suffit de comprendre que tout accès dans un tableau peut générer une erreur. Par exemple, il n'est pas possible d'accéder à `getRegion(3,6)`. Si ce cas de figure se produit, alors il n'est pas possible de continuer l'exécution normale du programme. L'objet tableau réagit en émettant une exception; "la cellule (5,6) n'existe pas". Si cette exception est levée, alors la clause $catch$ est activée

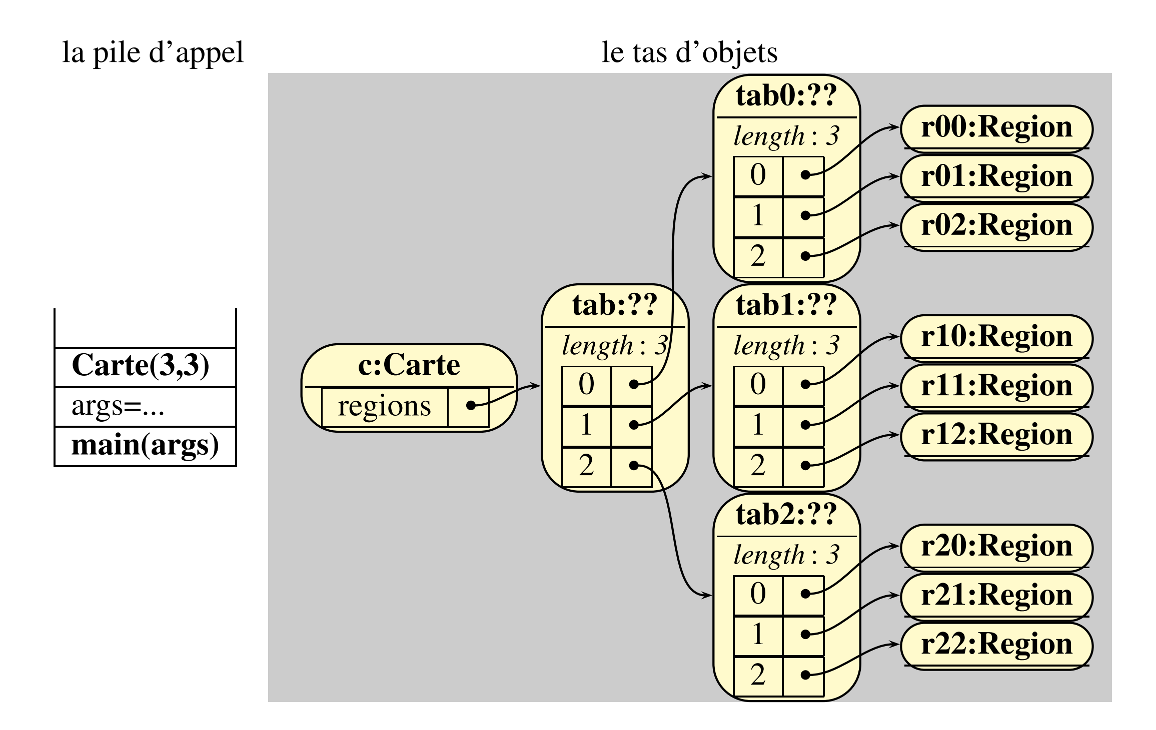


FIG. 3.61 – *Vue mémoire d'un objet tableau à deux dimensions*

et dans notre cas, la méthode se termine en renvoyant $null$. Il faut donc faire attention en utilisant la méthode $getRegion$ car elle renvoie soit la region demandée si elle existe, $null$ sinon. C'est à l'appelant de prendre en compte cette particularité. Il est alors extrémement utile de commenter son code pour avertir tout utilisateur de cette méthode de son mode de fonctionnement.

Nous avons parlé d'objets tableau. Mais comme nous sommes dans une langage de classe, si il y a objet, il y a une classe permettant d'instancier ces objets. Ou est la classe "tableau à deux dimensions de regions" ? Cette classe est en fait générée par le compilateur quand il rencontre des déclarations de tableaux. Ces classes ne sont pas visibles directement par le programmeur. Il est possible de les observer en utilisant la dynamicité du langage. Nous reviendrons sur ce point ultérieurement !!!

La classe $Region$ est tout à fait classique et ne pose aucune difficulté. Il faut toutefois remarquer comment le constructeur de cette classe maintient une référence sur l'objet carte. Ce comportement du constructeur est là pour représenter la relation $Carte - Region$; une carte est formée de $n * m$ regions, une région appartient à une carte. La relation de carte vers region est implantée un tableau car elle est multi-evaluée. La relation inverse est juste représentée par une référence classique. Toutefois comme la cardinalité de la relation inverse est de 1, i.e. toute région est située dans une carte, cela signifie qu'un objet région ne peut exister sans un objet carte. Donc pour construire une région, il faut une carte. Donc forcément, le constructeur de $Region$ doit prendre une référence sur un objet $carte$ en paramètre.

La figure 3.63 montre comment nous avons réellement implanté le diagramme logique de la figure **??**.

```
class Region {
    // relation inverse de la Carte−Region (1−1)
    private Carte carte ;
    private int x;
    private int y;

    // relation Carte−Mauhly (0−1) dans ce sens
    private Mauhly mauhly=null;

    public Region(Carte carte , int x, int y) {
        this . carte =carte ;
        this .x=x;
        this .y=y;
    }

    public Carte getCarte () {
        return carte ;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void setMauhly(Mauhly mauhly) {
        this .mauhly=mauhly;
    }

    public Mauhly getMauhly() {
        return mauhly;
    }
}
```

FIG. 3.62 – *relation 1-1 et 0-1*

3.11 Exemple recapitulatif

Nous allons maintenant utiliser dans un programme très simple toutes les notions que nous vues jusqu'à présent. Le programme en question va mettre en place une carte 5x5, et créer 10 montres. Les monstres vont alors se déplacer et combattre selon une stratégie prédéfinie pendant 10 tours (cf figure 3.64).

La carte est gérée comme décrit dans la figure 3.63. C'est donc la carte qui crée les régions. La logique du jeu lui-même est intimement lié au comportement des monstres et se trouve fort logiquement dans la classe Mauhly. Nous commentons maintenant le code de cette classe.

```
class Mauhly {
```

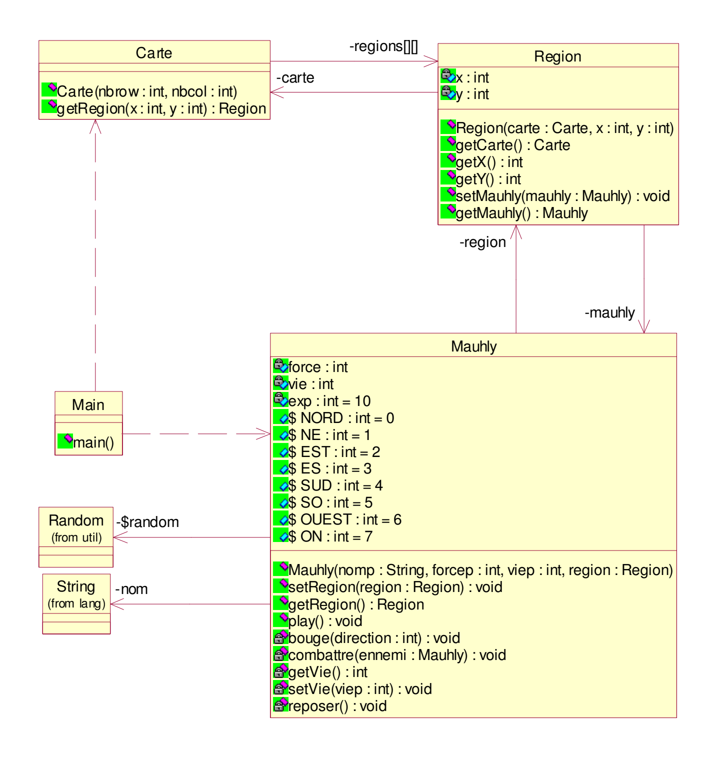


FIG. 3.63 – *Diagramme de classe physique pour la gestion de la carte*

```
class Main {
    public static void main(String args []) {
        Carte carte =new Carte (5,5);
        Mauhly monstres[]=new Mauhly[5];
        for (int i=0;i<monstres. length ; i }}) {
            monstres[ i ]=new Mauhly("momo" }i,10,10,carte.getRegion(i , i ));
        }

        for (int i=0;i<10;i }}) {
            for (int j=0;j<monstres. length ; j }}) {
                monstres[ j ]. play ();
            }
        }
    }
}
```

FIG. 3.64 – *Programme principal*

```
private String nom;
private int force;
private int vie;
private int exp=10;
private Region region;
...
```

La partie ci-dessus décrit l'état d'un monstre de classe Mauhly comme nous le décrivons depuis le début de ce chapitre. Il faut bien remarquer la référence $region$ représentant le fait qu'un monstre est toujours situé sur une région. C'est l'implantation de la relation Maulhy/Region dans le sens $Maulhy$ vers $Region$.

```
public static final int NORD=0;
public static final int NE=1;
public static final int EST=2;
public static final int ES=3;
public static final int SUD=4;
public static final int SO=5;
public static final int OUEST=6;
public static final int ON=7;

private static java.util.Random random=new java.util.Random();
```

Cette partie illustre une utilisation classique des constantes en Java. Nous utilisons ici, des constantes pour symbolyser les differentes direction dans lesquelles les monstres peuvent se déplacer. La définition de ces constantes permet par la suite de manipuler des directions et non des entiers.

La dernière ligne permet de créer un générateur de nombres aléatoires. La notation $java.util.Random$ permet de référencer la classe $Random$ située dans le package $java.util$. Attention, la classe s'appelle bien $java.util.Random$ et non $Random$, ce qui est une source courrante d'erreur de compilation en Java.

Le générateur de nombre aléatoire est bien un objet référencé par une variable de classe. Ceci est une construction tout à fait normale. L'utilisation d'une variable de classe est ici tout à fait pertinente. Le générateur est partagé par toutes les instances de la classe $Mauhly$. Il est initiliasé quand la classe $Mauhly$ est chargée par le chargeur de classes et toutes les instances demanderont un nombre aléatoire au même générateur ce qui le rend plus efficace.

```
public Mauhly(String nomp, int forcep, int viep, Region region) {
    nom=nomp;
    force=forcep;
    vie=viep;
    this.region=region;
    region.setMauhly(this);
}

public void setRegion(Region region) {
    this.region=region;
}

public Region getRegion() {
    return region;
}
```

```
public int getVie() {
  return vie ;
}
private void setVie(int viep) {
  vie=viep;
}
public void reposer() {
  setVie(getVie()+10);
}
```

LA partie ci-dessus est tout à fait classique, elle définit les différents accesseurs et modificateur de la classe $Mauhly$. Cette partie est quasiment automatiquement dérivée de la description de l'état de la classe.

```
public void play() {
    bouge(random.nextInt(this.ON));
}
```

La méthode $play$ permet de faire jouer le monstre. Ici, le monstre bouge dans une direction au hasard. Nous aurions pu coder une stratégie plus élaborée en tenant de la position des autres monstres et de l'état physique du monstre courant. Mais cette stratégie est suffisante pour ce que nous voulons montrer.

```
private void bouge(int direction) {
    int x=getRegion().getX();
    int y=getRegion().getY();

    switch (direction) {
    case NORD: {y--; break;}
    case NE: {x}};y--;break;}
    case EST: {x}}; break;}
    case ES: {x}}; y}}; break;}
    case SUD: {y}}; break;}
    case SO: {x--; y}}; break;}
    case OUEST: {x--; break;}
    case ON:{ x--;y--;break;}
    default: {
        System.err.println("Mauvaise direction ! (″} direction }″)″);
    }
    }
    // x,y représente maintenant les coordonnées de la region arrivée.
    Region regionCible=this.getRegion().getCarte().getRegion(x,y);

    if (regionCible==null) {
        return;
    }

    if (regionCible.getMauhly()==null) {
        this.setRegion(regionCible);
        regionCible.setMauhly(this);
    } else {
        combattre(regionCible.getMauhly());
    }
}
```

La méthode *bouge* est le coeur du compertement des monstres. Elle commence par déterminer en fonction d'une direction la région d'arrivée du monstre (*regionCible*). La définition des constantes est ici payante car elle permet une bonne lisibilité du code. Remplacez les constantes par 0,1,...,7 et le code devient beaucoup plus difficile à suivre.

La ligne 64 montre bien comment il est possible de naviger dans le graphe d'objet (à condition d'avoir bien conçu son programme). Ici, nous montrons bien comment nous remontons à partir de l'instance courrante sur la région et sur la carte pour finir par trouver la région cible.

Cette région peut ne pas exister auquel cas, le monstre reste juste ou il est et son tour est passé.

La ligne 70 teste si la région cible contient un monstre. Si ce n'est pas le cas, le monstre de déplace sur la région cible et libère sa région d'origine. Il suffit de mettre à jour la relation Mauhly/Region pour réaliser cette opération. Si la région cible contient un monstre alors il faut combattre, le monstre ne se déplace pas dans ce cas.

```
    private void combattre(Mauhly ennemi) {
        ennemi.setVie(ennemi.getVie()-this.force);
        System.out.println(this.nom}"␣inflige␣"}this.force} "pts␣de␣degat␣a␣"}ennemi.nom);
    }
}
```

La méthode de combat est très simple. Elle peut être déclarée private, car elle ne fait partie de l'interface de la classe. Dans le corps de cette méthode *this* frappe et *ennemi* reçoit et donc perd de la *vie*. Il faut remarquer que nous passons bien par l'interface de l'objet *ennemi*. Comme *ennemi* est de la même classe que *this* nous aurions pu accéder directement les variables d'instance de *ennemi*. Mais bien sur dans ce cas, nous aurions mal appliqué le principe d'encapsulation.

4

L’héritage

Nous avons vu que pour créer un objet il faut définir sa classe. Cette classe peut d'ailleurs servir à créer plusieurs objets : plusieurs instances. Différents constructeurs et leurs paramètres permettent même de faire varier les objets créés.

Mais comment crée-t-on des classes, sans trop se compliquer la vie ?

La solution classique est de construire un classe à partir d'une autre en lui ajoutant, ou en modifiant des méthodes et attributs. C'est le mécanisme d'héritage.

Notons tout de suite qu'en Java l'héritage est **simple** : une classe se définit explicitement à partir d'au plus une classe. Ainsi les classes forment des arbres d'héritage.

4.1 Exemple

Voyons tout de suite ceci sur un exemple. Introduisons une nouvelle classe pour notre jeu :

– La classe `Equipement` : un objet qui peut être mis dans le sac. Un équipement posséde une masse pour supporter de futures règles (limitation de la masse totale du sac). Il possède aussi un prix pour faire l'inventaire en fin de partie et supporter une extention possible des règles sur le calcul du score.

À partir de cette classe, trois nouvelles classes peuvent être définies :

– `Protection` : un équipement avec en plus un coefficient de protection (résistance).
– `Arme` : un équipement avec un coefficient d'efficassité (dégat)
– `Potion` : un équipement qui peut s'appliquer sur un monstre ou au joueur.

On dit que les classes `Arme`, `Protection` et `Potion` **héritent** ou **dérivent** de la classe `Equipement`. On dit aussi qu'elles sont **sous classes** de `Equipement`. Réciproquement on dit que la classe `Equipement` est **super classe** des classes `Arme`, `Protection` et `Potion`.

Le mécanisme d'héritage peut s'enchaîner plus d'une fois pour construire de nouvelles classe à partir d'une classe elle même définie par héritage. Nous pouvons ainsi introduire dans notre exemple :

– Deux formes de `Protections` : `Bouclier` et `Armure`.
– Trois types d'`Armes` : `Massue`, `Glaive`, `Hache`.
– Deux types de `Potions` : `PotionDeVie`, `PotionParalysante`.

Pour l'instant ces sept dernières classes sont juste une simple classification : aucun nouvel attribut ou méthode n'est ajouté pour les définir.

Donnons la notation UML associée à la définition des classe introduites par héritage. Les flèches avec une extrémité triangulaire blanche se lisent *A hérite de B* si A est la source de la flèche.

Le diagramme de classe UML de la figure 4.1 donne une idée de cette notation pour notre exemple.

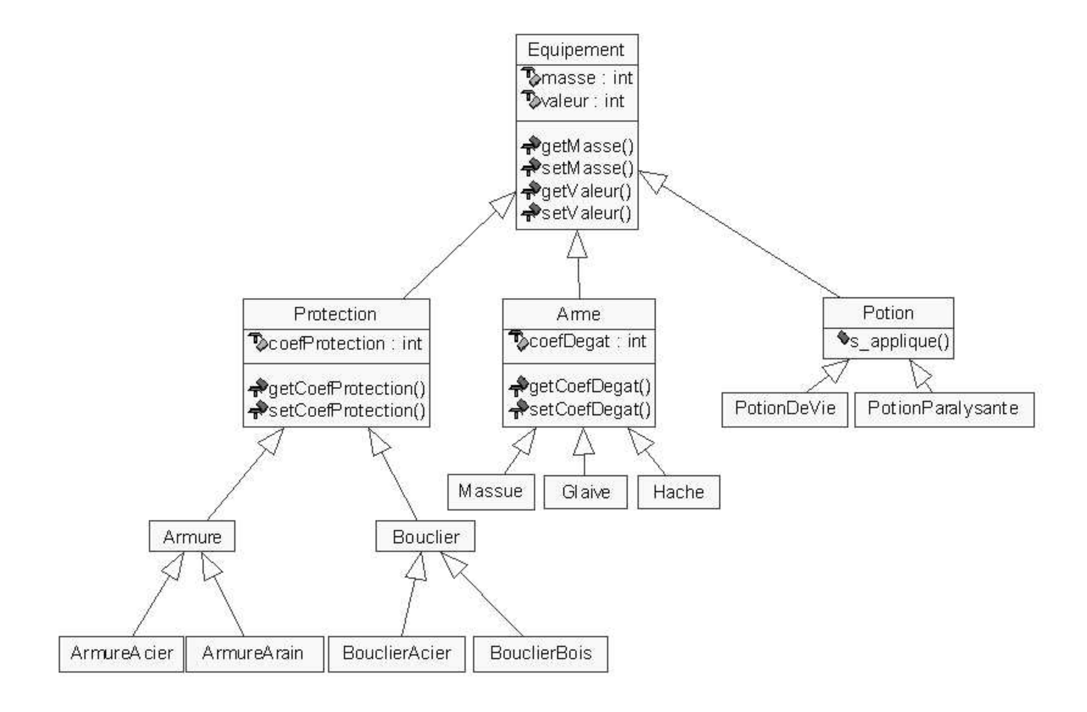


FIG. 4.1 – *Diagramme de classe pour les équipements d'un joueur*

4.2 Syntaxe en Java

En Java pour définir une classe par héritage il faut utiliser le mot clef **extends**. En effet une sous classe étend (et modifie) la définition de sa super classe.

Pour la classe `Equipement` de laquelle dérivent les classes `Protection` et `Armure` dérivant de nous obtenons respectivement le code des figures : Fig. 4.1, Fig. 4.2 et Fig. 4.3 Dans ce dernier exemple on constate que le corps de la classe est vide ce qui correspond bien à la remarque faite dans la section précédente : elle n'introduit rien de nouveau, sauf son nom par rapport à sa super classe.

Figure 4.1 Code de la classe `Equipement`

```
public class Equipement {
    int masse;
    int valeur;

    int getMasse() {return masse;}
    void setMasse( int masse ) {this.masse=masse;}

    int getValeur() {return valeur;}
    void setValeur( int valeur ) {this.valeur=valeur;}
}
```

Figure 4.2 La classe `Protection` dérivée de `Equipement`

```
public class Protection extends Equipement {
    int coefProtection ;
    int getCoefProtection (){ return coefProtection ;}
    void setCoefProtection ( int coef ){
        this . coefProtection =coef;
    }
}
```

Figure 4.3 La classe `Armure` dérivée de `Protection`

```
public class Armure extends Protection {}
```

4.3 Relation avec les éléments constituant une classe

Par défaut une sous classe est formée de tout ce qu'elle définit et de tout ce que définissent ses super classes.

Dans les cas des exemples de la section 4.1, la classe `Armure` hérite de `Protection` qui hérite elle même de `Equipement`, et donc elle possède les attributs `coefProtection`, `masse`, `valeur`, et toutes les méthodes associées à ces attributs (`getMasse`, `getCoefProtection`...).

Notons tout de suite qu'en Java toute classe hérite par défaut de la classe `Object` (cf. §4.7).

Passons maintenant en revue les intéractions entre les éléments constituant une classe (cf. chapitre 3) et le mécanisme d'héritage.

4.3.1 Héritage et méthodes d'instance

Lors de l'héritage si la sous classe définit une méthode d'instance de **même nom** et même liste de types des paramètres formel qu'une de celle héritée : on dit que la première méthode **surcharge** dernière. Le terme anglais utilisé dans la documentation Java est : *overriding*. Dans ce cas l'invocation de la méthode sur une instance de la sous classe conduit **obligatoirement** à l'exécution de cette nouvelle méthode définie dans la méthode.

Reprenons l'exemple de la classe `Equipement` et de ses sous classes. Nous y introduisons Fig. 4.4 pour des fins de mise au point une méthode `void print()` qui affiche les caractéristiques d'un équipement c'est à dire sa masse et sa valeur.

Figure 4.4 La classe `Equipement` avec une méthode `print`

```
public class Equipement {
    int masse;
    int valeur;

    int getMasse() {return masse;}
    void setMasse( int masse ) {this.masse=masse;}

    int getValeur() {return valeur;}
    void setValeur( int valeur ) {this.valeur=valeur;}
    void print() {
        System.out.println("Equipement␣:␣masse="+
                           getMasse()+",␣valeur="+
                           getValeur());
    }
}
```

Si nous ne modifions pas les sous classes (`Arme`, `Protection`, ...) lorsque la méthode `print()` est appelée sur une instance de l'une d'entre elles c'est la méthode `print()` qui vient d'être définie dans `Equipement` qui sera exécutée. En d'autres termes la méthode `print()` ainsi définie sera celle appelée pour toute instance d'objet créée par un : `new Equipement()`, `new Protection()`, ou `new Arme()` : le **type dynamique** de l'instance considérée.

Ce n'est pas faux, mais on pourrait être plus précis. On peut par exemple surcharger la méthode `print()` de la classe `Protection` afin de faire apparaître en plus les propriétés propres aux protections (`coefProtection`). La classe ainsi obtenue est présentée sur la figure Fig. 4.5.

Figure 4.5 Surcharge de `print` dans la classe`Protection` sous classe d'`Equipement`

```
public class Protection extends Equipement {
    int coefProtection;

    int getCoefProtection(){return coefProtection;}
    void setCoefProtection( int coef ){
        this.coefProtection=coef;
    }
    void print() {
        System.out.println(
              "Protection␣masse="+
              getMasse()+",␣valeur="+
              getValeur()+
              ",␣coefficient␣de␣protection="+
              getCoefProtection());
    }
}
```

Cette fois c'est la méthode `print()` ainsi définie qui sera appelée pour tout instance

```
Equipement : masse=0, valeur=0
Protection masse=0, valeur=0, coefficient de protection=0
Protection masse=0, valeur=0, coefficient de protection=0
Protection masse=0, valeur=0, coefficient de protection=0
```

FIG. 4.2 – *Exécution de l'exemple de la figure 4.6*

de `Protection` ou d'une de ses sous-classes. Notons que nous nous efforçons de toujours utiliser les accesseurs même lorsque nous surchargeons une méthode.

Le mécanisme impliqué dans le choix de ma méthode à exécuter s'appelle **liaison dynamique**. Pourquoi un tel non ? Souvent la méthode exacte à appeler pour une instance repérée par une référence ne peut pas être définie lors de la compilation, mais seulement à l'exécution lorsque l'on est sûr du **type dynamique** de l'instance d'objet considéré. Schématiquement il faut qu'un `new` ai eu lieu pour que l'on puisse déterminer la classe exacte de l'objet et donc accéder à la "bonne" méthode. Le choix de la méthode s'effectue du bas vers le haut dans l'arbre d'héritage à partir du type dynamique. On considère d'abord cette classe puis éventuellement ses super classes pour trouver une méthode de même nom et signature.

Illustrons ceci dans un petit programme Java donné dans la figure Fig. 4.6. Le résultat de l'exécution de ce programme est donné dans la figure 4.2. Sur la ligne 3 de ce programme est créée une instance de `Equipement`, le type dynamique de l'objet référencé par la variable `e1` est donc `Equipement`. De la même façon est créée sur la ligne 4 une instance de `Protection` référencée par la variable `e2`. La ligne 5 conduit donc à l'exécution de la méthode définie dans la classe `Equipement` alors que les lignes 6, 8 et 9 conduisent à celle la méthode définie dans la classe `Protection`. En effet l'affectation (parfaitement licite (cf. §**??**)) dans une référence de type statique différent (ligne 7) ou un *cast* (ligne 9) vers une super classe ne change en aucun cas le type dynamique de l'occurrence considérée qui reste **le point départ** pour rechercher la méthode à exécuter dans l'arbre d'héritage.

Figure 4.6 Utilisation d'une méthode surchargée

```
public class Test_print {
    public static void main(String [] args){
        Equipement e1 = new Equipement();
        Protection e2 = new Protection ();
        e1. print ();
        e2. print ();
        e1=e2;
        e1. print ();
        ((Equipement)e2). print ();
    }
}
```

La liaison dynamique est incontournable et toujours présente pour l'appel d'une méthode d'instance. Il y a un moyen en java de modifier le point de départ de la recherche de méthode dans l'arbre d'héritage du type dynamique : la référence prédéfinie `super`. Cette référence «spéciale », du même ordre que la référence **this**, permet de référencer dans le code **d'une**

```
void print() {
        <super>.print();
        System.out.println(getValeur()+
                ", coefficient de protection="+
                getCoefProtection());
}
```

FIG. 4.3 – *Exemple d'utilisation de* ***super***

méthode d'instance ou d'un constructeur l'objet courant, (**this**) comme un objet ayant pour type dynamique la super classe du type dynamique de `this`. Nous reviendrons en détail sur ce mécanisme dans le chapitre 5.

Illustrons ceci de nouveau par notre méthode `print`. Actuellement (cf. Fig. 4.5) nous avons écrit la méthode `print` de la classe `Protection` en y recopiant le code de la méthode `print` (lignes 7 à 12) de `Equipement` et en ajoutant du code réellement relatif aux attributs de `Protection` (lignes 13 et 14). On a finalement écrit deux fois le code de la méthode `print` de `Equipement`. On a envie d'appeler la méthode `print` de la super classe lors de la définition de la méthode `print` de la classe `Equipement`. Ceci est parfaitement possible et s'écrit :

4.3.2 Méthodes de classe

Le rapport des méthodes de classes (statiques) avec l'héritage est très différent du cas des méthodes d'instance puisqu'il n'y a pas d'instances à considérer !

Si une sous classe (disons `B`) définit une méthode de classe (disons `m`) avec les même nom et profil qu'une autre méthode de classe de l'une de ses super classe (disons `A`), on dit que la méthode définie dans `B` **masque** celle définie dans `A`. Le terme anglais utilisé dans les documentations Java pour le masquage est *hiding*.

Il faut noter qu'en Java, c'est une erreur à la compilation de vouloir masquer (par une méthode statique) une méthode d'instance. La réciproque est aussi une erreur à la compilation il est interdit de vouloir surcharger une méthode de classe par une méthode d'instance.

Quel effet a ce masquage ? Et bien dans le code de `B` (code d'initialisation de variable ou bloc statique, méthode d'instance ou de classe) l'appel `m(....)` sera toujours celui de la méthode définie dans B. Sans artifice particulier l'accès à la méthode définie dans la super classe reste impossible, la méthode `m` de `A` est bien "masquée" par celle de `B`. `B` n'hérite pas de la méthode définie dans `A`.

Il est possible d'accéder à une méthode masquée par :

- la notation *<nom de classe>.<nom de méthode>(...)*. Cette notation permet d'invoquer la méthode de la classe indiquée à partir du code de n'importe quelle autre classe.
- la notation *<expression retournant une référence à un objet>.<nom de méthode>*. Dans ce cas la méthode invoquée sera celle du type **statique** de l'expression utilisée: une classe déterminée lors de la **compilation**.
- la référence prédéfinie **super**. Dans le cas du code d'une méthode **d'instance**, d'un

constructeur, du code d'initialisation d'une variable **d'instance** ou d'un bloc anonyme d'initialisation non static d'une sous classe, cette référence permet d'invoquer les méthodes de classe de la super classe. Dans ce cas finalement seule le type statique associé à **super** à la compilation nous intéresse, l'objet référencé par super (le même que **this**) ne sera pas accédé à l'exécution). **super** permet donc d'accéder à une méthode statique à partir de tout comme lié à une instance.

Le programme de la figure 4.7 et son exécution de la figure 4.4 mettent en valeur qu'un appel à une méthode statique, masquée ou non est résolu **statiquement à la compilation**. Cet exemple n'est pas lié au jeu, car finalement on masque assez rarement. Les lignes 1 à 5 de cet exemple définissent une classe `A` contenant juste une méthode de classe `m` qui affiche *A.m()*. À partir de la ligne 6 est définie la classe `B` sous classe de `A`. Elle présente (lignes 7 à 9) une méthode de classe `m` qui masque celle de `A` et affiche *B.m()*. `B` possède une méthode de classe `test` (lignes 10 à 11) qui fait un appel simple à `m`. Elle possède aussi une méthode **d'instance** : `test2` (lignes 13 à 15) qui appelle la méthode `m` via la référence **super**. Passons au programme de la classe `Main` : une instance de `A` référencée par une variable de type statique `A` est crée en ligne 19. La même chose est faite à la ligne 20 pour une instance de `B` et une variable de type statique `B`. La ligne 21 invoque explicitement la méthode `m` de `A` (ligne 1 de la figure 4.4). La ligne 22 invoque explicitement celle de `B` (ligne 2 de la figure 4.4). La ligne 23 invoque la méthode `m` de `A` (ligne 3 de la figure 4.4) car le type statique de la variable `a` est `A`. Par contre c'est celle de `B` qui est accédée en ligne 24 (ligne 4 de la figure 4.4) pour des raisons similaires. La ligne 25 appelle en fait la méthode `m` de `B` puisque la méthode `test` appartient bien au code de `B` et donc la notation `m()` utilisée en ligne 11 accède à la méthode de `B` masquant celle de `A`. Ce n'est plus le cas de la ligne 26 car la méthode `test2` utilise la référence **super** (ligne 14) qui a finalement comme type statique : `A`. On constate que la dernière ligne de l'exécution du programme correspond à un appel à la méthode `m` de `A` car la ligne 28 du programme de test est **syntaxiquement** la même que la ligne 23 et donc l'affection de la ligne 27 n'a aucune influence puisqu'elle ne change pas le type statique de la variable `a`.

A.m()
B.m()
A.m()
B.m()
B.m()
A.m()
A.m()

FIG. 4.4 – *Exécution de l'exemple de la figure 4.7*

Figure 4.7 Exemple de masquage d'une méthode de classe et utilisation

```
class A {
    public static void m(){
        System.out. println ("A.m()");
    }
}
class B extends A {
    public static void m(){
        System.out. println ("B.m()");
    }
    public static void test (){
        m();
    }
    public void test2 (){
        super.m();
    }
}
class Main {
    public static void main(String [] args){
        A a= new A();
        B b= new B();
        A.m();
        B.m();
        a.m();
        b.m();
        b. test ();
        b. test2 ();
        a=b; // permutation
        a.m();
    }
}
```

Le masquage est un comportement foncièrement différent de la liaison dynamique, mais peu comparable puisqu'il ne concerne pas les instances.

4.3.3 Héritage et attributs

Pour les attributs (variables de classe ou d'instance) c'est un mécanisme de **masquage** similaire à celui des méthodes de classe (cf. §4.3.2) qui s'applique.

Il faut bien noter que dans les deux cas : variables de classe et variables **d'instance** la sélection de l'attribut à utiliser se fait de manière **statique à la compilation**.

4.3.4 Héritage et constructeurs

Nous avons vu que l'état initial d'une instance est défini lors de l'appel d'un constructeur. Les constructeurs peuvent être définis pour chaque classe qu'elle soit sous classe d'une autre ou pas (en fait nous verrons que de toute façon une classe est au moins sous classe de la classe `Object`. Les constructeurs bien que proche des méthodes d'instances **ne s'héritent pas**.

Par contre à l'exécution il y a un shéma strict d'appel des constructeurs disponibles le long de l'arbre d'héritage.

L'arbre d'héritage est d'abord remonté jusqu'à la classe `Object`, puis les constructeurs appelés en commençant par celui de `Object`. C'est donc une succession d'appel de la racine de l'arbre d'héritage vers la feuille qu'est par exemple la classe indiquée sur un `new` (construction du haut vers le bas).

Dans sur la première ligne d'un constructeur **et seulement là** on peut utiliser le nom de méthode prédéfinie : `super(...)`. Cet appel, selon le type des paramètres effectifs permet de sélectionner l'appel d'un constructeur de la super classe **immédiate**.

Si la première ligne d'un constructeur n'est ni un appel à `this` et ni un appel à `super` le compilateur introduit implicitement sur la première ligne un appel à `super()` : appel au constructeur sans paramètre de la super classe.

Considérons la hiérarchie de classes et le programme de test de la figure 4.8.

Figure 4.8 Héritage et constructeurs

```
class A1 {
    int a=10;
    A1(){
        System.out. println ("CODE␣2;␣a="+a);
    }
}
class B extends A1 {
    int b=5;
    B(){
        System.out. println ("CODE␣3;␣b="+b);
    }
    B( int i ){
        this ();
        b=i;
        System.out. println ("CODE␣4;␣b="+b);
    }
}
class Main {
    public static void main( String [] args ){
        B unB=new B(100);
    }
}
```

Que ce passe-t-il exactement lors de l'exécution de l'expression `new B(1000)` (ligne 20) ?

– `new` réserve l'espace mémoire nécessaire pour les attributs définis dans `B` et toutes ses super classes.
– L'appel effectif fait dans `main` exécute le code du constructeur de `B` avec un paramètre entier (lignes 12 à 16).
– La première ligne de code de ce constructeur (`this()` (ligne 13) appelle le constructeur sans paramètre de `B` (lignes 9 à 11).
– La première ligne de code (ligne 10) du constructeur sans paramètre de `B` n'est ni un appel à `this(...)` ni un appel à `super(...)`, donc un appel à `super()` a été introduit par le compilateur. Ceci conduit donc à l'appel du constructeur (sans paramètre) de la super classe de `B` : `A1` (lignes 3 à 5).
– Dans le constructeur de `A1` (ligne 3) la même chose se répète et conduit à l'appel du constructeur sans paramètre de la classe `Object`. Bien entendu ce constructeur de cette classe un peu particulière ne fait pas d'appel à `super()` puisque cette classe n'a pas de super classe.
– Les variables d'instance introduites par la classe `Object` sont initialisées à l'aide des affectations indiquées dans leur déclaration, puis le reste du constructeur de la classe `Object` s'exécute et retourne.
– Les variables d'instance introduites par la classe `A1` sont initialisées à l'aide des affectations indiquées dans leur déclaration (ici `a=10;` ligne 2), puis le reste du constructeur de la classe `A1` continue (ligne 4) et retourne.

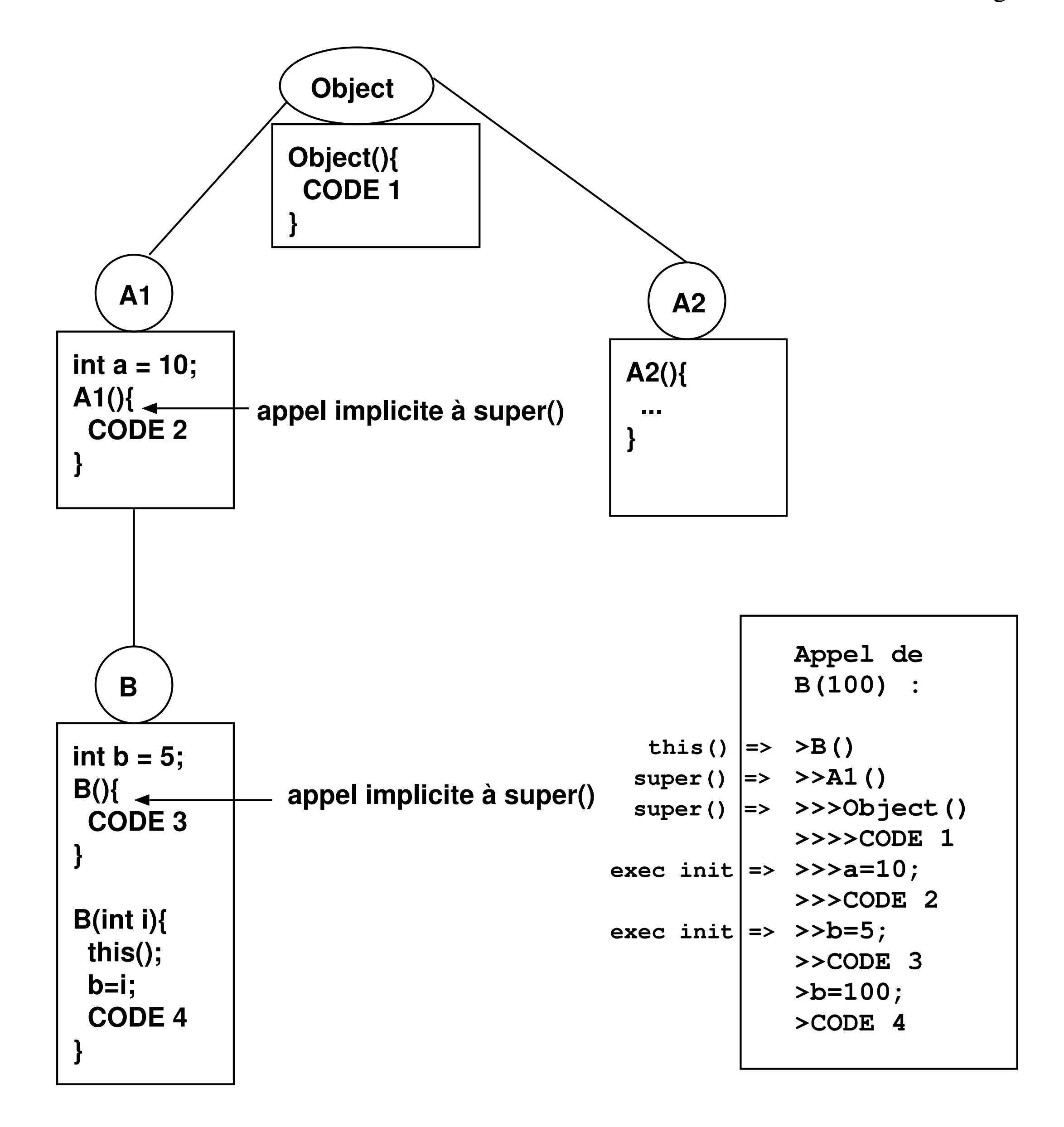


FIG. 4.5 – *Exemple succession d'appels de constructeurs*

- Les variables d'instance introduites par la classe `B` sont initialisées à l'aide des affectations indiquées dans leur déclaration (ici `b=5;` ligne 8), puis le reste du constructeur sans paramètre de la classe `B` continue (ligne 10) et retourne.
- Le constructeur avec un paramètre de la classe `B` continue (lignes 14 et 15) et retourne.
- L'appel `new B(100)` est enfin terminé.

La figure Fig. 4.5 résume cette succession d'appels.

L'insertion par défaut de `super()` dans un constructeur d'une classe est faite par le lors de la compilation, cela revient **exactement à écrire soi-même** `super()`, et donc **la super classe doit posséder un constructeur sans paramètre**. Nous avons vu que le contructeur par

défaut (sans paramètre) est créé lui aussi automatiquement par le compilateur **si aucun autre constructeur n'est explicitement défini par le programmeur**. Une erreur de compilation classique survient souvent lorsque qu'un ou plusieurs constructeurs avec paramètres sont définis dans une classe, et qu'un constructeur d'une classe est dans la situation d'insertion de `super()` par défaut.

4.3.5 Remarques diverses

Type du résultat d'une méthode surchargée

Il faut noter que le type du résultat d'une méthode (ou `void` le cas échéant) n'est pas pris en compte pour différencier les signatures des méthodes afin de déterminer s'il y a surcharge ou masquage. Par conséquent si une méthode surcharge ou masque une autre elle doit avoir le même type de résultat que la méthode de la super classe. Dans le cas contraire une erreur de compilation est générée.

4.4 Classes et méthodes abstraites

4.4.1 Méthodes abstraites

Certaines méthodes d'une classe peuvent juste avoir leur profil de spécifié (nom, type des paramètres, type du résultat), mais **sans bloc de code associé** (corps). Une telle méthode *abstraite* devra être implementée dans une sous classe. Pour déclarer une telle méthode en java il suffit de faire précéder son nom et profil du mot clef **abstract** et de ne mettre qu'un `;` en lieu et place du corps de la fonction.

Ceci s'applique aussi bien à des méthodes d'instance qu'à des méthodes de classe.

Voyons ceci un exemple sur une évolution de la classe `Equipement` présenté sur la figure 4.9 Nous avons rendu abtraites les méthodes relatives à la valeur et à la masse d'un équipement (lignes 2 et 3). En effet ses caractéristiques peuvent varier selon l'usage de l'objet pendant le jeu, et donc il faut connaître plus précisément la nature d'un objet pour connaître sa masse et sa valeur.

Il faut bien noter que si une classe possède au moins une méthode abstraite elle doit être elle aussi déclarée être comme abstraite. Ceci s'écrit en java en faisant précéder le mot clef `class` du mot clef `abstract` lors de la définition de la classe.

Figure 4.9 Un exemple de classe abstraite

```
public abstract class Equipement {
    abstract int getMasse();
    abstract void setMasse( int masse );

    abstract int getValeur ();
    abstract void setValeur ( int valeur );

    public String toString () {
        return "Equipement␣:␣masse="+getMasse()+
                ",␣valeur="+getValeur ();
    }
    void print () {System.out. println ( this . toString ());}
}
```

La figure 4.10 donne une implantation effective des méthodes abstraites de la classe `Equipement` pour une `Potion`. Ceci est fait en surchargeant (lignes 6, 9, 10 et 13) les méthodes déclarées comme abstraites dans l'exemple précédent. Dans ce cas la masse de l'équipement, une fiole de potion varie selon l'état l'objet : la variable d'instance `bue` (ligne 11).

Cette classe `Potion` implemente donc “pour de bon” les méthodes relatives à la masse et à la valeur d'une potion, selon qu'elle ai été bue ou non. Dans `Potion` toutes les méthodes abstraites **héritées** de `Equipement` ont été implémentées donc cette classe n'a plus besoin d'être elle même déclarée comme `abstract`.

Figure 4.10 Implémentation des méthodes abstraites de la figure 4.9

```
public class Potion extends Equipement {
    private final static int MASSE_FIOLE=30;// grammes
    private final static int MASSE_LIQUIDE=5; //grammes
    int valeur; // valeur ... si non bue !
    boolean bue=false;
    int getMasse(){
        return MASSE_FIOLE+(bue?0:MASSE_LIQUIDE);
    }
    void setMasse( int masse ){};
    int getValeur (){
        return bue?0: valeur;
    }
    void setValeur ( int valeur ){
        this . valeur=valeur;
    }
    public String toString () {
        return "Potion␣:␣bue?="+bue+
                "␣(␣"+ super. toString ()+")";
    }
}
```

Une classe abstraite peut très bien être utilisée pour effectuer des opérations tout à fait

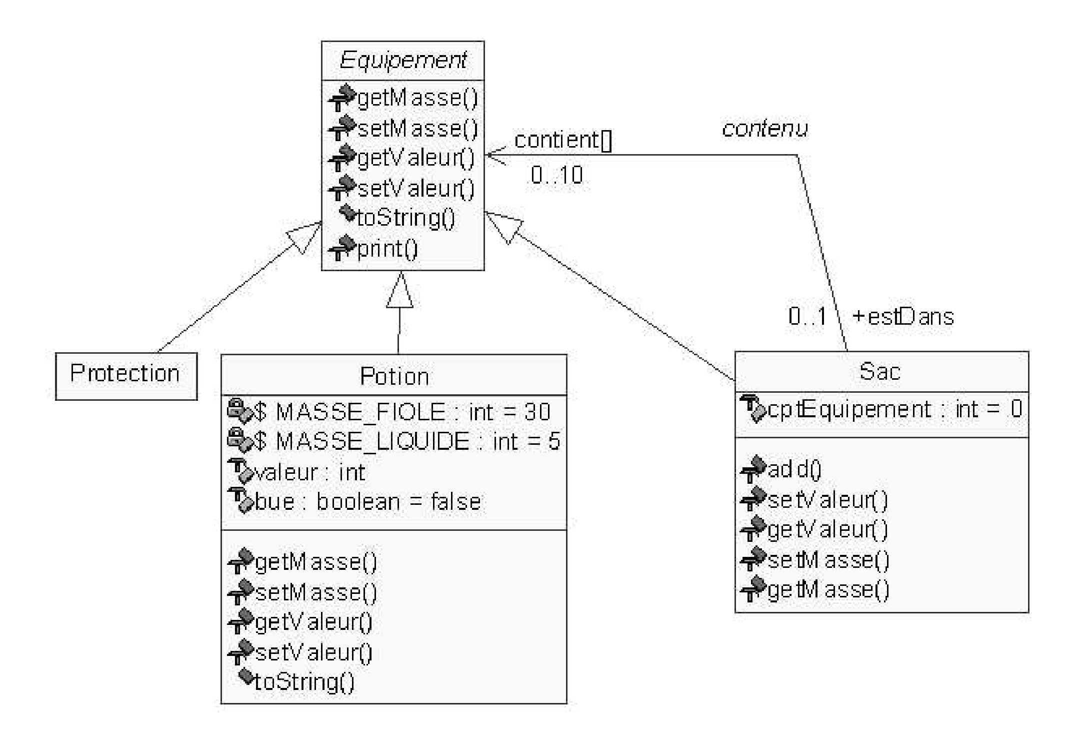


FIG. 4.6 – *Diagramme de classe d'un exemple de* pattern composite

Participant du *pattern*	classe java de l'exemple
Component	`Equipement`
Composite	`Sac`
Leaf	`Protection`, `Potion` et sous classes

TAB. 4.1 – *Correspondance entre les composants du* pattern composite *et la classe* `Sac`

concrètes. Introduisons un nouvel équipement un peu spécial : un sac pouvant contenir des équipements.

Un sac étant lui aussi un équipement, sa valeur et sa masse sont à calculer. Elles le seront à partir des méthodes déclarées comme abstraites dans la classe `Equipement` qu'il peut appeller sur les éléments qu'il contient. La classe `Sac` a donc à faire à un ensemble polymorphe d'équipements (`Armure`, `PotionDeVie`, ...) mais dont il fait un usage homogène grâce aux méthodes abstraites déclarées au niveau de la classe `Equipement`.

Le diagramme de classe ULM de la figure 4.1 est donc un peu étendu par le diagramme de la figure 6.2).

Notre sac est l'application du *pattern composite (163)* [**?**]. Utilisé avec la correspondance indiquée dans la table 7.1.

Le code java correspondant est donné dans la figure 4.11. De manière assez naturelle la classe `Sac` n'est pas déclarée comme abstraite : un sac est un équipement parfaitement défini et utilisable en temps que tel. Les méthodes abstraites de la classe `Equipement` y

sont complétement implémentées. Il faut par contre remarquer qu'un sac perçoit les objets qu'il contient comme de simples objets de la classe `Equipement` : les éléments du tableau `contient` (ligne 2). Bien entendu les types dynamiques des éléments de ce tableaux seront des sous classes non abstraites de `Equipement` (`PotionDeVie`, `Armure`, ...). Pour par exemple calculer la masse total d'un sac (lignes 14 à 18) il suffit de savoir que ce sont des équipements disposant d'une méthode `getMasse`, et à l'exécution la liaison dynamique permet d'accéder à la méthode `getMasse` correcte selon le type dynamique de l'élément considéré. Il faut aussi noter que notre sac actuel n'a ni masse ni valeur propre.

Une classe `Sac` "*réaliste*" devrait avoir au moins une méthode de plus : `remove(Equipement)` pour supprimer un élément contenu dans le `Sac`.

Figure 4.11 Utilisation de méthodes abstraites et implémentation du *pattern Composite*

```
public class Sac extends Equipement {
    Equipement[] contient =new Equipement[10];
    int cptEquipement=0;
    void add(Equipement e){
        if (cptEquipement<contient.length){
            contient [cptEquipement]=e;
            cptEquipement++;
        } else {
            // exception ...
        }
    }
    void setValeur( int valeur ){}
    int getValeur (){
        int resultat =0;
        for( int i=0; i<cptEquipement; i++){
            resultat +=contient[ i ]. getValeur ();
        }
        return resultat ;
    }
    void setMasse( int masse ){}
    int getMasse(){
        int resultat =0;
        for( int i=0; i<cptEquipement; i++){
            resultat +=contient[ i ]. getMasse();
        }
        return resultat ;
    }
}
```

4.4.2 Classes abstraites et problème d'instanciation

Une erreur classique est de penser d'une classe abstraite n'est pas instanciable. Bien entendu si `A` est une classe abstraite, l'opérateur **new** ne peut pas lui être appliqué. Mais si `B` est une classe non abstraite dérivée de `A` toute instance de `B` peut aussi être vue comme une instance de `A`, l'opérateur binaire **instanceof** renvoie **true** dans les cas (cf. §**??**).

L’astuce préconisé par [**?**][§8.6.8] pour qu’aucune instance ne puisse être dérivée à partir d’une classe est de définir un unique constructeur **private** pour celle-ci. L’opérateur **new** ne peut alors pas être utilisé de l’extérieur et aucune sous classe ne peut être définie car ses éventuels constructeurs ne pourraient faire appel à ce constructeur privé (cf. §4.6.1). Ceci n’empêche pas la classe de créer **elle-même** des instances et de transmettre leurs références vers l’extérieur, dans des méthodes de classes. Pour être encore plus restrictif la classe peut aussi être déclarée comme abstraite, même si elle ne possède pas de méthodes abstraites (qui ne pourraient d’ailleurs pas être implémentées dans les sous classes!). Ainsi l’utilisation de **new** sera aussi impossible dans le code de la classe elle-même. L’exemple de la figure 4.12 applique ces deux astuces à une classe `A` (lignes 1 et 2). La figure 4.7 donne des erreurs de compilations obtenues lorque l’on essaie de dériver d’instancier la classe abstraite `A` même à partir d’elle-même (lignes 1 à 3) ou lorsque l’on essaie de la dériver en une classe `B`(lignes lignes 5 à 7).

On peut se poser la question de l’”utilité d’une telle chose. Certaines classes sont parfois formées uniquement de variables et méthodes statiques : typiquement des fonctions s’appliquant sur des types primitifs (un ensemble de fonctions mathématique sur des réels par exemple). Dans ce cas, créer une instance créer une instance de la classe ne rime à rien et il vaut mieux empêcher ceci par tous les moyens.

Figure 4.12 Une classe totalement non instanciable

```
abstract class A {
    private A() {
        System.out. println ("A");
    }
    static public A factory () {
        return new A();
    }
}
class B extends A {
    B() {
        System.out. println ("B");
    }
}
class TestNonInstanciable {
    public static void main(String [] args){
        A a=A.factory ();
    }
}
```

4.5 Classes, méthodes et attributs “finaux”

4.5.1 Classes finales

La déclaration d’une classe java peut être précédée du mot clef : **final**, ceci a pour effet de rendre impossible toute définition de nouvelles sous classes à partir de cette classe.

```
TestNonInstanciable.java:6: class A is an abstract
class. It can't be instantiated.
        return new A();
               ^
TestNonInstanciable.java:10: No constructor
matching A() found in class A.
    B() {
    ^
2 errors
```

FIG. 4.7 – *Erreurs de compilation lors de l'usage d'une classe non instanciable*

Une classe "finale" ne peut donc jamais être utilisée à droite du mot clef **extends**. Ses méthodes de classes ne seront jamais masquées et ses méthodes d'instance jamais surchargées.

La classe `Sac` (cf. §4.4.1) pourrait être définie comme finale afin que ses méthodes ne puissent pas être surchargée et que personne ne puisse changer l'implementation choisie pour le contenu du sac.

4.5.2 Méthodes finales

Si la définition d'une méthode d'instance ou de classe est précédée du mot clef **final** elle ne pourra pas être surchargée (resp. masquée) dans une sous classe.

Il faut noter que toute méthode déclarée dans une classe **final** est **implicitement final**. On peut la déclarer explicitement comme **final**, ceci est redondant mais autorisé. Un constructeur ne peut pas être déclaré comme **final**.

On pourrait empêcher toute modification de l'implantation du contenu d'un `Sac` en déclarant comme finales toutes les méthodes actuelles de la classe `Sac`, sans la déclarer elle-même comme finale. Elle pourrait être ainsi sous classer pour ajouter des propriétés relatives à un type de sac particulier, mais en empêcher de remettre en cause (par surcharge) ce qui est déjà implementé.

4.5.3 Attributs finaux

L'influence de **final** sur les variables d'instances ou de classes a été présenté dans la section **??**. Il n'a aucune influence sur les possibilités de **masquage**. Une variable d'instance ou de classe déclarée comme finale peut parfaitement être masquée dans une sous classe.

4.6 De l'influence de `protected`, `public`, et autres contrôle d'accès

4.6.1 Influence du mot clef private

L'utilisation du mot clef **private** (cf. §3) devant une déclaration de méthode ou attribut de classe ou d'instance rend inaccessible cet élément à tout code excepté celui de sa classe.

Du point de vue de l'héritage cela veut dire que les attributs et méthodes déclarés comme **private** ne sont pas hérités par les sous classes. Il n'y a **pas de liaison dynamique** dans ce cas.

Le mot clef **private** peut aussi être appliqué à la définition d'un constructeur. Dans ce cas son influence est réduite à l'accès puisque les constructeurs ne s'héritent pas (cf. §4.3.4). L'utilisation de **private** dans ce cas empêche l'appel de **new** pour le constructeur concerné à partir de code extérieur à sa classe. Elle empêche aussi l'utilisation de la notation **super(....)** associée à ce constructeur dans les constructeurs des classes immédiatement dérivées. La génération d'appel par défaut à **super()** (cf. §4.3.4) peut interférer avec cette dernière restriction d'accès et générer de subtiles erreurs à la compilation. Par contre l'utilisation de **new** (resp. **super(...)**) reste parfaitement utilisable dans le code de la classe (resp. dans les constructeurs de la classe).

La figure 4.13 illustre ce cas. On y reprend la classe `Protection` mais l'attribut `coefProtection` est maintenant défini comme **private** afin qu'il ne puisse être modifié qu'au travers des méthodes *set, get* associées. La classe `Armure` est aussi reprise en ajoutant une méthode `reparer` qui relève de deux le coefficient de protection directement et par la référence **super** mais sans passer par les méthodes d'accès. La figure 4.8 donne les erreurs obtenues lors de la compilation de la classe `Armure`. La première erreur de la figure 4.8. Elle montre bien que la variable `coefProtection` n'existe pas dans la classe dans la classe lorsqu'on essaie de l'utiliser à la ligne 10 de la figure 4.13. La deuxième erreur correspond à l'utilisation de la référence **super** faite à la ligne 11 de l'example java. La notation utilisée permet bien de désigner la variable `coefProtection` de la classe `Equipement` mais pas d'y accéder puisqu'elle est déclarée comme **private** (ligne 2 de l'exemple java).

Figure 4.13 private et héritage

```
public class Protection extends Equipement {
    private int coefProtection ;
    int getCoefProtection (){ return coefProtection ;}
    void setCoefProtection ( int coef ){
        this . coefProtection =coef;
    }
}
public class Armure extends Protection {
    void reparer (){
        coefProtection +=10;
        super. coefProtection +=10;
    }
}
```

4.6.2 Influence du mot clef protected

L'idée qu'il faut avoir du mot clef **protected** utilisé devant la déclaration d'une méthode d'instance (constructeur compris) ou de classe, ou celle d'une variable d'instance ou de classe est que l'élément ainsi déclaré ne peut être accédé que par la classe elle-même ou une de ses sous classe. La définition exacte donnée dans [**?**][§6.6.2] est compliquée. Il vaut mieux

```
Armure.java:3: Undefined variable: coefProtection
            coefProtection +=10;
            ^
Armure.java:4: Variable coefProtection in class
      Protection not accessible from class Armure.
            super. coefProtection +=10;
                  ^
2 errors
```

FIG. 4.8 – *Erreurs de compilation de la classe* `Armure` *de la figure 4.13*

garder en tête l'idée présentée ici, qui est une légère restriction de la définition, on évite ainsi certaines erreurs subtiles.

En fait **protected** est une protection selon une "*encapsulation de classe*", le code d'une instance d'une classe peut très bien accéder à des méthodes ou variables d'instances d'une autre instance de la même classe ou d'une sous classe.

la notation `super(...)` permet d'appeler un constructeur protégé, par contreUtilisation de **new** sur un constructeur **protected** n'est pas autorisé.

Nous verrons au chapitre 7 que lorsque rien n'est précisé lors de la déclaration d'un attribut ou d'une méthode, l'accés par défaut est "*package*". Celui-ci prime sur **protected**.

4.7 La classe `Object`

Comme déjà évoqué plusieurs fois, il existe une classe "tout en haut de la hiérarchie". Lorsque que l'on écrit :

```
class A {....}
```

cela revient à :

```
class A extends Object {...}
```

Cette classe `Object` est prédéfinie par java et vient avec le *package* `java.lang` (cf. §**??**). Elle forme donc une racine commune à tous les arbres d'héritage entre classes que l'on peut former en Java.

Cette classe fournit des méthodes liées à l'implantation des instances et au problèmes de synchronisation lorsque que plusieurs activités (*threads*) accèdent à un même objet.

Certaines méthodes peuvent être surchargée dans les sous classes.

Une méthode très souvent surchargée à des fin de mise au point est la méthode `toString`. Elle est sensée renvoyer une représentation textuelle concise mais représentative de l'objet (dixit la documentation du jdk).

Dans le cas de la classe `Equipement` et ses sous classes, il suffit de reprendre ce qui a été proposé pour les méthodes `print` Fig. 4.14. Grâce à la liaison dynamique, toutes les méthodes utilisant `toString`, comme les méthodes affichant les erreurs et les exceptions, utiliseront la méthode `toString` ainsi spécialisée. C'est aussi le cas de la méthode `print`

de Equipement qui a été réécrite (simplifiée) et qui n'a plus à être surchargée dans chaque sous classe, la surcharge de toString suffit. La figure 4.15 illustre ce dernier point pour la classe Protection.

Figure 4.14 Surcharge de la méthode toString de Object

```
public class Equipement {
    int masse;
    int valeur;
    int getMasse() {return masse;}
    void setMasse( int masse ) {this.masse=masse;}
    int getValeur() {return valeur;}
    void setValeur( int valeur ){this.valeur=valeur;}
    public String toString() {
        return "Equipement␣:␣masse="+
               getMasse()+",␣valeur="+
               getValeur();
    }
    final void print() {
        System.out.println(this.toString());
    }
}
```

Figure 4.15 Utilisation de la surcharge de la méthode toString de Object

```
public class Protection extends Equipement {
    int coefProtection;
    int getCoefProtection(){return coefProtection;}
    void setCoefProtection( int coef ){
        this.coefProtection=coef;
    }
    public String toString() {
        return "Protection␣:␣"+ getValeur()+
            ",␣coefficient␣de␣protection="+
            getCoefProtection()+
            "␣("+super.toString()+")";
    }
}
```

Les méthodes de synchronisations (wait, notify, notifyAll correspondent aux accès à un moniteur lié à l'instance considérée, ne doivent en aucun cas être modifiées, elles sont d'ailleurs définies comme final (cf. §4.5.2).

Une autre méthode **final** de la classe Object est la méthode getClass qui retourne la classe, c'est à dire le **type dynamique** de l'instance considérée. Pour être précis cette méthode retourne une instance de la classe Class qui contient toute la description du type. Récuper la classe d'une instance peut être utile à des fins de mise au point, mais aussi pour utiliser des mécanismes sophistiqués d'appels dynamiques de méthodes. Il est tout à fait justicieux que cette méthode soit déclarée comme **final** dans la classe Object : ainsi aucune

classe, qui dérive forcément de la classe `Object` ne pourra surcharger `getClass` pour tenter de faire prendre ses instances pour ce qu'elles ne sont pas !

4.8 Questions

Voici quelques questions pour voir si les différents concepts de ce chapitre ont été compris.

1. La classe `Object` possède-t-elle un constructeur sans paramètres, si oui pourquoi ?
2. Une méthode peut-elle être déclarée comme abtraite et privée ?
3. Une classe peut-elle être déclarée **final** et **abstract** ?
4. Une méthode peut-elle être déclarée comme abstraite et finale ?

4.8.1 Petits exercices

Q : Ecrire une classe la plus simple possible. La compiler. Visualiser le contenu du .class généré à l'aide de `javap`. Que dire ?
R :

– on écrit `classe A` dans un fichier A.java. On compile `javac A.java`
– `javap A` donne :

```
class A extend java.lang.Object {
        A();
}
```

On observe donc bien :

– Toute classe hérite de `Object`
– Un constructeur sans paramètre est généré par défaut.

5
La liaison dynamique

Nous avons déjà évoqué le mécanisme de liaison dynamique (cf. §4.3.1) commun aux languages orientés objet. Grossièrement une notation codeo.m(....) où o est une référence à un objet dont le type **statique** déclaré est C, et m une méthode d'instance de la classe C, n'appelle (invoquation) pas forcément la méthode de C à l'exécution mais plutôt une méthode "plus proche" du type dynamique de l'objet référence, une sous classe de C.

Ce mécanisme est la base du polymorphisme et de nombreux *design patterns*.

Sa mise en œuvre peut avoir des effets inattendus. Nous allons préciser dans ce chapitre quels points essentiels pour éviter certaines erreurs.

5.1 Mécanismes mis en œuvre

Nous allons uniquement considérer le cas de l'invocation d'une méthode d'instance **non déclarée comme private**, pouvant donc être soumise à une surcharge paramétrique et à une surcharge par héritage. Cette présentation est une adaptation de [**?**][§15.11, pages 323-343].

Trois formes d'invocation peuvent se présenter :

1. <nom de méthode>.(<expression1>, <expressions2>, ...) Cette forme doit être contenue dans du code relation à une instance (méthode, expression d'initialisation d'une variable d'instance, bloc non statique d'initialisation.
2. <expression>.<nom de méthode>(<expression1>, <expressions2>, ...) Expression doit retourner une référence vers une classe ou une interface.
3. **super**.(<expression1>, <expressions2>, ...) Une telle forme ne peut pas être utilisée dans la classe Object (cf. §4.7) et l'utilisation de this doit être valide dans le même contexte.

5.1.1 Actions à la compilation

Trois "étapes" (*steps*) sont effectées par le compilateur lorsqu'il considère cette invocation de méthode.

1. **Classe ou interface à partir de laquelle la méthode est cherchée**. C'est :
   - Forme 1 : la classe contenant le code de l'invocation.
   - Forme 2 : la classe ou interface type de <expression>.
   - Forme 3 : la super-classe de la classe contenant le code de l'invocation.
2. **Détermination de la signature de la méthode** (types des paramètres formel) à appeler. Cette recherche s'effectue dans la classe ou l'interface déterminée à l'étape précédente. Toutes les méthodes héritées par celle-ci sont considérées par le compilateur. Cette étape peut être décomposée en plusieurs sous-étapes :

   (a) Le compilateur sélectionne toutes les déclarations de méthodes accessibles (selon leur déclaration **protected**, **private** ou visibilité *package*) depuis la classe ou l'interface *applicables* au cas de l'invocation considérée : - même nombre de paramètres, types statiques des <expression1,2..> des paramètres effectifs affectables aux types des paramètres formels (*method invocation conversion*). Une restriction s'applique ici : **en aucun cas le compilateur restreint un paramètre effectif entier** (int) vers un paramètre formel plus petit (byte, short, char).

(b) Parmi toutes les méthodes applicables le compilateur en choisi une seule dont la signature sera conservée dans le code. Le critère utilisé est que les types paramètres de paramètres formels soient les “plus proches” de ceux des expressions utilisées comme paramètres effectifs. La spécification du langage parle de *spécificité* d'une déclaration. Une déclaration de méthode provenant d'une classe ou interface `C1` est dite plus spécifique qu'une autre provenant d'une classe ou interface `C2` si et seulement si : - `C1` peut être convertie (au sens de la conversion utilisé au point précédent) et si chaque type de paramètre de la méthode applicable provenant de `C1` peut être converti (même convertion que précédemment) en le type du paramètre de même rang de la méthode provennant de `C2`. Cette relation de spécificité est **une relation d'ordre partielle** entre déclarations de méthodes. Si le compilateur ne peut pas trouver une déclaration “minimun” au sens de cette relation une erreur de compilation est générée : l'invocation de méthode est ambigue.

Il faut bien noter que le type du résultat de la méthode n'est pas pris en compte dans cette étape. La méthode est sélectionnée **indépendamment** du type de son résultat, des erreurs de compilations liées à un problème de typage du résultat de l'invocation dans l'expression qui l'englobe peuvent donc se produire. Nous verrons une exemple dans la section suivante.

3. **Vérification que la méthode choisie est appropriée** Nous nous sommes placé dans le cas de l'invocation d'une méthode d'instance. Mais syntaxiquement rien ne différencie une invocation de méthode d'instance de celle d'une méthode de classe. Il faut donc dans le cas général vérifier que la méthode sélectionnée après les deux étapes précédente à un sens dans le contexte de l'appel. Il faut vérifier entre autre que si la méthode est une méthode d'instance l'invocation a bien lieu dans un contexte où **this** a un sens.

Il faut bien noter que ces trois étapes statiques ont une influence sur le **code généré** pour l'invocation considérée. Il faut avoir en tête que la **signature exacte** de la méthode cherchée est conservée dans le code (parmi d'autre informations). Les pré-traitements faits par le compilateur rendre plus simple et efficasse la recherche de ma méthode à appeler lors de l'exécution (*method dispatch*).

5.1.2 Liaison à l'exécution

Voyons maintenant comment est évalué le code généré pour notre expression d'invocation, en utilisant bien entendant les informations pré-calculées dans la section précédente.

Là aussi plusieurs étapes sont effectuées :

1. **Détermination de la référence d'objet** sur laquelle a lieu l'invocation. Dans les cas des formes syntaxiques 1 et 3 cette référence est simplement `this` : objet concerné par le bloc de code en cours d'exécution. `this` a forcément un sens à cet endroit car cela a été vérifié à l'exécution. Dans le cas restant : la référence est le résultat de l'évaluation de `<expression>`.
2. **Évaluation des arguments** Les expresions `<expression1, 2, 3...>` sont évaluées dans cet ordre, de gauche à droite.

3. **Vérification d'accès à la méthode** `m` en fonction du code code faisant l'invocation et de la déclaration de la méthode (`public`, `protected`, ...).
4. **Localisation de la méthode à invoquer**. C'est la liaison dynamique proprement dite puisque que nous nous sommes placé dans le cas d'une méthode d'instance non privée. La méthode `m` à appeler va être recherchée dans l'arbre d'héritage à partir d'un point qui dépend de la forme de l'invocation.

   – Formes 1 et 2 : recherche à partir du type dynamique de l'objet référence par la référence calculée en 1. C'est la classe utilisée lors de l'appel à `new` utilisé pour créer cet objet. Il faut bien faire attention, même si la référence considérée est `this`, le type dynamique peut très bien être celui d'une sous classe de la classe du code où `this` est utilisé.
   – Forme 3 : recherche à partir de la **super-classe** de la classe qui contient le code de l'invocation.

   Une fois cette classe de départ déterminée, la méthode `m` avec la signature et le type de résultat calculé statiquement (cf. §5.1.1) y est recherchée. Si elle n'y est pas trouvée, cette recherche est répétée à partir de la super-classe
   Si la référence calculée au point 1 est nulle (`null`) une erreur à l'exécution est générée (exception `NullPointerException`).

5.2 Quelques exemples d'erreurs liées aux mécanismes d'invocation

5.2.1 Utilisation des types statiques lors de la sélection des signatures

Voyons un effet lié au point des mécanismes d'invocation d'une méthode. Nous utilisons volontaire un exemple sans signification pour bien mettre en valeur le problème. Considérons le programme de la figure 5.1. Nous y definissons deux classes `A` et `B` juste pour avoir deux possibilité de typage. La classe `Test` possède deux méthode d'instance de même nom l'une avec la signature `A` et l'autre avec la signature `B` (lignes 6 et 9). Le programme principal crée une instance de `Test` pour pouvoir appeler les méthodes précédentes (ligne 13). Il crée aussi une instance de `B` référencée par une variable de type statique `B` (ligne 14). Cette instance est aussi référencée par une variable de type statique `A` super-classe de `B` (ligne 15). Les deux invocations de méthodes des lignes 16 et 17 bien qu'elles ressemblent beaucoup : même nom et même instance en paramètre ne font pas appellent aux mêmes méthode. La recherche des méthodes possible s'effectue à partir de la classe `Test`, type statique de l'expression `t`. La première correspond à la méthode de la la ligne 6 : le type statique du de l'expression en paramètre effectif (`a`) est `A`, elle a donc l'unique signature valide, le compilateur ne peut affecter `A` vers `B` (réduction ou*narrowing*) afin de sélectionner l'autre méthode. Le cas de la deuxième invocation est un peu différent. `B` peut être affectée vers `A` (sous-classe vers super-classe, *widening*) les signatures des deux méthodes peut être sélectionnées. Par contre pour la même raison que précédemment la méthode de la ligne 9 est plus spécifique que celle de la ligne 6, c'est donc elle qui est réellement appelée. La figure 5.1 donne le résultat de l'exécution de cet exemple. On y voit clairement que les deux méthodes sont appelées.

A
B

FIG. 5.1 – *Exécution de l'exemple 5.1*

Figure 5.1 Détermination de la signature d'une invocation et type statiques des paramètres effectifs

```
class A {
}
class B extends A {
}
class Test {
    void m(A a){
        System.out. println ("A");
    }
    void m(B b){
        System.out. println ("B");
    }
    static public void main(String [] args){
        Test t=new Test();
        B b=new B();
        A a=b;
        t.m(a);
        t.m(b);
    }
}
```

5.2.2 Informations générées à la compilation et modification ultérieurs des classes

Java est supposé être pratique pour mettre en œuvres de librairies (*packages*) indépendants (cf. §**??**) distribués par différentes producteurs ou vendeurs. En effet une modification dans un package si elle ne modifie pas les accès publiques de celui-ci peut se faire sans modification ou recompilation des codes des classes clientes extérieures au package. Mais les mécanismes décrits dans la section précédente peuvent introduire de subtils problèmes à l'exécution pas réellement "naturels" pour le programmeur.

Explication du problème et effets

La signature de la méthode à chercher lors d'une liaison dynamique est conservée dans le code généré pour une expression d'invocation donc souvent dans le code d'une classe cliente de la classe qui possède la méthode visée. En cas de modifications de cette dernière le code généré pour l'invocation ne donc aucune raison pour être modifier.

Illustrons ceci sur un exemple, simplification extrême de l'application du *pattern visitor* présenté à la section 6.7. Considérons les classes et le programme de test de la figure

```
Arme.accept
 visiter Equipement
```

FIG. 5.2 – *Exécution de l'exemple 5.2*

5.2. L'exécution du programme `Test` est donnée sur la figure 5.2. Cette exécution n'a rien d'étonnante : la ligne 22 est bien l'invocation de la méthode d'instance `accept` de la classe `Arme` puisque la référence `a` contient bien une référence vers l'objet de type dynamique `Arme` créé à la ligne 21. La méthode `accept` concernée fait un appel à la méthode d'instance `visiter` de la classe `Visiteur` (ligne 15). La (seule) méthode sélectionnable pour cette invocation a pour signature `visiter(Equipement)`. Nous pouvons vérifier ceci en décompilant le code généré pour la classe `Arme` en utilisant la commande `javap` du jdk. Nous utilisons les options : -p pour ne considérer que les méthodes publiques et -c pour avoir un aperçu du p-code généré. Le résultat de l'appel `javap -public -c Arme` est donné sur la figure 5.3. On constate bien sur la ligne 17 de ce listing que l'invocation de `visiter` est bien celle d'une méthode d'instance (*invokevirtual*) et que la signature attendue possède un paramètre de type `Equipement`.

Figure 5.2

```
class Visiteur {
    public void visiter (Equipement e) {
        System.out.println(" visiter Equipement");
    }
}
class Equipement {
    public void accept( Visiteur v){
        System.out.println("Equipement.accept");
        v.visiter (this);
    }
}
class Arme extends Equipement {
    public void accept( Visiteur v){
        System.out.println("Arme.accept");
        v.visiter (this);
    }
}
class Test {
    public static void main(String [] args){
         Visiteur v=new Visiteur ();
        Equipement a=new Arme();
        a.accept(v);
    }
}
```

Modifions la classe `Visiteur` comme indiquée sur la figure 5.3, en surchargeant (surcharge paramétrique) la méthode `visiter` par une deuxième méthode `visiter` prendre une `Arme` en paramètre. Si nous recompilons juste la classe `Visiteur` (`javac Visiteur.java`)

```
Compiled from Arme.java
class Arme extends Equipement {
    public void accept( Visiteur );
}

Method Arme()
   0 aload_0
   1 invokespecial #7 <Method Equipement()>
   4 return

Method void accept( Visiteur )
   0 getstatic #8 <Field java.io.PrintStream out>
   3 ldc #1 <String "Arme.accept">
   5 invokevirtual #9 <Method void println(java.lang.String)>
   8 aload_1
   9 aload_0
  10 invokevirtual #10 <Method void visiter (Equipement)>
  13 return
```

FIG. 5.3 – *Décompilation de la classe* `Arme` *de l'exemple 5.2*

l'exécution du programme `Test` demeure celle de la figure 5.2. En effet le code généré pour la classe `Arme` n'a pas changée et donc la signature de la méthode `visiter` recherchée lors de la liaison dynamique est `visiter(Equipement)` même si la nouvelle méthode ajoutée est plus spécifique.

Pour que la nouvelle méthode soit prise en compte il faut recompiler les classes clientes : les classes `Equipement` et `Arme` dans notre cas. Comme indiquer précédemment cette solution peut ne pas être praticable, si par exemple la classe `Visiteur` est distribuée dans un package à part et que le reste de l'application est déjà déployée.

Figure 5.3 Surcharge d'une méthode de visiteur

```
class Visiteur {
    public void visiter (Equipement e) {
        System.out.println("visiter␣Equipement");
    }
    public void visiter (Arme a) {
        System.out.println("visiter␣Arme");
    }

}
```

Code de "défense"

Une solution "défensive" pour contourner le problème précédent est conseillée dans [**?**][page 331]. Elle permet qu'un méthode nouvellement surchargée soit bien prise en compte sans recompilation des classes clients. Il suffit de modifier aussi la méthode existante afin de

rediriger les invocations concernant la nouvelle. Pour la classe `Visiteur` cette modification est indiquée sur la figure 5.4. Dans la méthode initiale qui continue d'être appelée il suffit de tester le type dynamique du paramètre (ligne 2) et d'appeler la nouvelle méthode surchargée (ligne 4). Il faut noter l'utilisation du transtypage explicite vers `Arme` de façon que le type (statique) du paramètre de cette deuxième invocation soit bien `Arme` afin que la nouvelle méthode soit bien applicable. `Arme` étant une sous classe de `Equipement` elle sera donc aussi la plus spécifique. Cette modification a donc bien l'effet escompté : si la méthode initiale sur les `Equipement` est appelée avec un paramètre donc le type dynamique est `Arme`, l'appel sera propagé à la nouvelle méthode dédiées à la visite des `Arme`.

Figure 5.4 Redirection d'invocation d'une méthode

```
class Visiteur {
    public void visiter (Equipement e) {
        if (e instanceof Arme) {
            visiter ((Arme)e);
        } else {
            System.out. println (" visiter ␣Equipement");
        }
    }
    public void visiter (Arme a) {
        System.out. println (" visiter ␣Arme");
    }
}
```

5.3 Liaison dynamique et constructeurs

Si la liaison dynamique n'a pas de sens sur les constructeurs (cf. §4.3.4), il faut tout de même noter que la liaison dynamique s'applique pour les appels de méthodes d'instance faits dans le code des constructeurs. Ce n'est pas le cas de C++. Ceci est tout à fait homgène et utile, mais peut succiter quelques effets inattendus. Notamment, il est ainsi possible d'accéder à des variables d'instances munies d'expression d'initialisation **avant** que celle-ci aient été exécutée. C'est pour cela que java définit des valeurs par défaut selon leur type (0 pour les numériques, `null` pour les références y compris celles sur la classe `String`).

Reprenons dans la figure 5.5 l'exemple illustrant l'enchaînement des appels des constructeurs vu à la section 4.3.4. Nous l'avons simplifié en supprimant un des constructeurs de `B`. Par contre nous avons introduit une méthode d'instance `getVal` d'accès aux variables d'instance définies dans `A` ou `B`. Cette méthode est donc surchargéee par héritage dans la classe `B` (ligne 14). La figure **??** donne le résultat de l'exécution de cet exemple. La première ligne correspondant à l'exécution du constructeur de `A`, en effet après remontée des constructeurs des super-classes, les appels effectifs se font du haut vers le bas (cf. §4.3.4). La première valeur affichée vaut 0, car c'est le résultat de l'appel de `getVal` de `B` en effet le type dynamique de `this` est `B` (ligne 19). Or lorsque que cette méthode est invoqué l'expression d'initialisation de la ligne 10 n'a pas encore été évaluée puisque la construction de la partie de l'objet liée à la sous classe se fera après celle liée à `B`. Donc la valeur de la variable d'instance `b` définie dans `B` est accédée par `getVal` **avant** d'être initialisée, la méthode accède donc à la valeur par défaut fixée par le langage : 0 dans la cas d'un nombre. Par contre le constructeur de `A` af-

getVal=0, a= 10
getVal=5, b= 5

FIG. 5.4 – *Exécution l'exemple 5.5*

fiche bien la valeur initialisée de la variable d'instance `a` puisque les expressions d'évalution d'une classe sont évaluées avant le constructeur. La deuxième ligne de la trace d'exécution conrespondant à l'exécution du constructeur de `B`. Cette fois l'appel à `getVal` de la ligne 12 bien qu'il accède toujours à la même méthode retourne 5 car l'expression d'initialisation de la ligne 10 a été évaluée avant l'appel du constructeur de `B`.

Ce ne n'est absolument pas le cas en C++. Si l'appel de méthode est pourtant bien déclarée comme virtual dans un constructeur, on appelle la méthode de la classe du constructeur, pas celle du type dynamique. De plus, si l'on veut forcer les choses à l'aide d'une méthode virtuelle pure (=0), on obtient une erreur à la compilation :

```
cons.c: In method `A::A()':
cons.c:8: abstract virtual `int A::getVal()' called from constructor
arc C 89 % a.out
```

Autrement, c'est vraiment impossible en C++.

Figure 5.5 Constructeurs et liaison dynamique

```
public class A {
    int a=10;
    A(){
        System.out. println ("getVal="+getVal()+", a= "+a);
    }
    int getVal (){ return a;}
}

public class B extends A {
    int b=5;
    B(){
        System.out. println ("getVal="+getVal()+", b= "+b);
    }
    int getVal (){ return b;}
}

public class Test {
    public static void main( String [] args ){
        B unB=new B();
    }
}
```

5.4 Conclusions

Bon Java est peut-être moins obscur que C++, car il y moins de fioritures (pas de : paramètres par défaut, nombres variables de paramètres, surcharges des opérateurs, héritage multiple, . . .), et donc moins *error prone* pour le programmeur. Mais il y toujours des mécanismes délicats qui peuvent générés des erreurs (compilation ou exécution) subtiles ou pire des effets très différents de ceux qu'attend le programmeur.

6

Les interfaces

6.1 Définition

Les interfaces sont une version réduites de définition de classe :

– forcément abtraites,
– pas de variable d'instance,
– pas de méthodes de classes (statiques),
– des variables forcément de classe (statiques) et finales : en d'autres termes **uniquement des définitions de constantes**,
– des méthodes d'instances forcément **abstraites**,
– des définitions forcément **publiques**.

On peut résumer cela en disant qu'une interface définie un *type abstrait* et les constantes s'y rapportant. Il n'est pas nécessaire de préciser **public final static** devant les déclarations de variables d'une interface ni **public abstract** devant les déclarations de méthodes d'une interface. Il est même recommandé de ne pas utiliser ces mots clefs qui sont en fait redondants vu la définition d'une interface.

6.2 Implementation d'une interface

En tant que telle une interface a un interêt très limité : elle ne permet pas en elle même de créer des "instances". Pour cela il faut qu'au moins une classe "implemente" cette interface : c'est à dire fournisse les même méthodes mais avec un corps. Le mot clef **implements** permet de relier une définition de classe à une ou plusieurs interface.

Une interface se compile comme une classe. Il est conseillé de créer un fichier `.java` (unité de compilation) du nom de l'interface. Comme pour les classes, l'usage demande que leurs noms commencent par une majuscule.

6.3 Références d'interface

Une référence d'objet peut être déclarée en utilisant le nom d'une interface en lieu et place du nom d'une classe. Une telle référence permet de référence **sans transtypage** (*cast*) n'importe quelle instance d'une classe qui implémente l'interface en question. Le contrôle de type à la compilation de Java utilise les liens d'implementation déclarés entre classes et interfaces.

Bien entendu seules les méthodes et variables définies dans l'interface utilisée à la déclaration de la référence sont accessibles via celle-ci.

Il est parfaitement correct de créer des tableaux de références à une interface. La syntaxe est la même de que celle vue précédemment (cf. §**??**) :

```
<nom d'interface>[] t=new <nom d'interface>[<nombre d'éléments]
```

Cette opération de construction de tableau, même si une interface est une classe abstraite, a bien un sens puisque seule des références sont créées, aucune instance d'objet n'est effectivement créée sauf le tableau lui-même.

L'accès aux constantes définies dans une interface peut aussi se faire par la notation :

```
<Nom d'interface>.<Nom de constante>
```

C'est la même notation que l'accès aux constantes d'une classe (cf. §**??**).

6.4 Notation en UML et exemple simple

Nous avons pour l'instant défini une petite hiérarchie d'equipements pouvent être mis dans un sac. Certains de ces équipements peuvent aussi être portés par le joueur, ils doivent en fait être obligatoirement être portés avant d'être utilisables : c'est le cas, pour l'instant des protections et des armes, voir du sac. Par contre, nous décidons que les les potions n'ont pas à être portées, elles sont simplement consommées, une seule fois. Ainsi le joueur n'aura pas à reposer son arme ou bouclier avant de pouvoir consommer une potion.

Nous pouvons introduire une interface spécifiant ce que doit être capable de faire un objet portable par le joueur sur différentes parties de son corps.

Du point de vue des notations UML cette interface peut s'introduire comme indiqué sur la figure 6.1. Les interfaces sont des "stéréotypes" UML basés sur les classes abstraites. Ce stéréotype est défini dans la spécification de base (*core package*) [**?**][page 2-36] de UML. Le fait qu'une classe fournit ce que spécifie une interface (**implements** en java) se représente par une association de réalisation où la classe est la source et l'interface la destination d'une flèche en pointillé avec un triagle blanc.

Un certain nombre de relations lient un joueur à des objets portables. L'interface `Portable` spécifie les méthodes pour accéder à ces relations côté objet porté. Elle fournit aussi des constantes pour désigner les emplacements (mains, dos, tibias ...) sur lequels un joueur peut porter un objet.

La définition en java l'interface `Portable` est représentée sur la figure **??**.

Figure 6.1 L'interface `Portable` en java

```
abstract interface Portable {
    public static int NON_PORTE=0;
    int MAIN_DROITE=1;
    int MAIN_GAUCHE=2;
    int DOS=3;
    Joueur getPorteur ();
    public abstract int getEmplacement();
    void porterMaintenant( Joueur j, int en );
}
```

Pour que le joueur puisse porter des armes à la main il faut que la classe `Arme` implemente cette interface, cela donne le code de la figure 6.2. Il faut aussi que la classe `Joueur` soient modifiée afin d'implanter les relations *porteEnMainDroite*, *porteEnMainGauche* et *porteSurLeDos* (voir le code de la figure **??**).

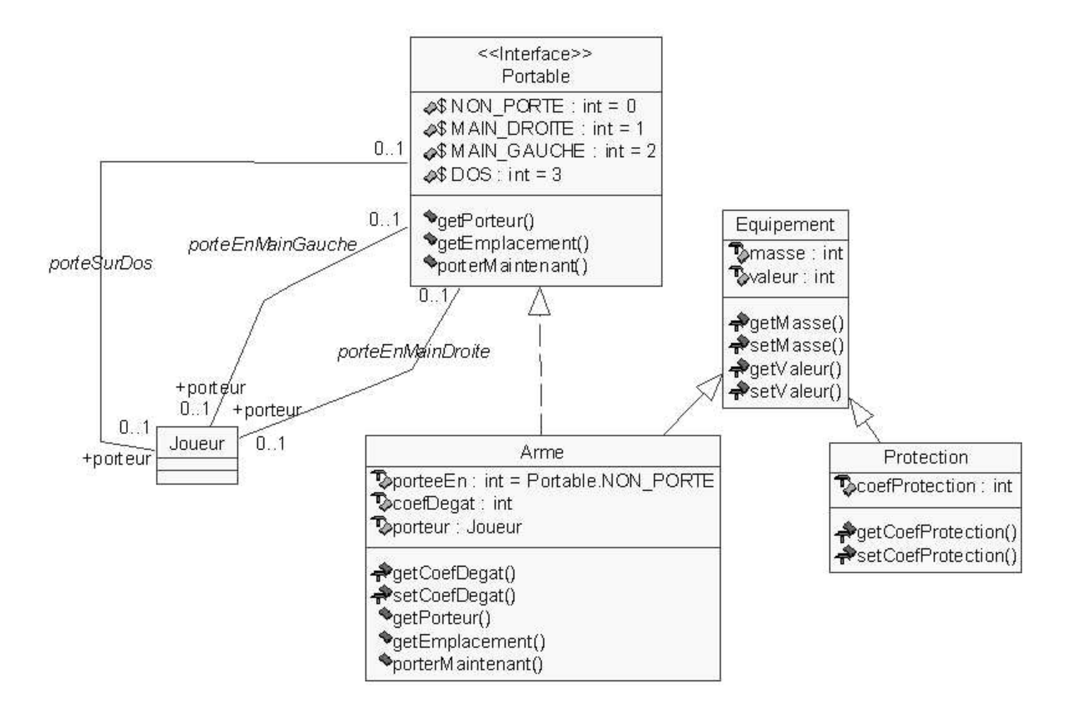


FIG. 6.1 – *L'interface* `Portable` *en UML*

Figure 6.2 Une implementation de l'interface `Portable`

```
public class Arme extends Equipement
implements Portable {
    Joueur porteur;
    int emplacement=Portable.NON_PORTE;
    int coefDegat;
    int getCoefDegat(){return coefDegat;}
    void setCoefDegat( int coef ){ this.coefDegat=coef;}
    // Ce que demande Portable.
    public Joueur getPorteur(){ return this.porteur;}
    public int getEmplacement() {return this.emplacement;}
    public void porterMaintenant( Joueur j, int en ){
        this.porteur=j;
        this.emplacement=en;
    }
}
```

Il faut bien noter que les ajouts faits dans la classe `Arme` devront être **répétés** dans la classe `Protection` ou `Sac` si on décide qu'elles doivent aussi être “portables”. Il faut entre autre que le sac puisse être porté sur le dos.

6.5 Interface, héritage, ambiguïté

6.5.1 Héritage mutiple des interfaces

Il est parfaitement possible de définir des hiérarchies d'interfaces. L'héritage **multiple** est disponible entre interfaces : le mot clef **extends** peut être suivi de plusieurs noms d'interface séparés par des virgules lors de la définition d'une interface. De même une classe peut implémenter plus d'une interface : le mot clef **implements** peut être suivi de plusieurs noms d'interface séparés par des virgules.

Finalement classes et interfaces forment un graphe **orienté** entre les relations d'héritage ou d'implementation (réalisation). On pourrait donc argumenter que finalement on dispose de l'héritage multiple en java. Ceci n'est pas réaliste, en tout état de cause le code les méthodes que possède une classe provient uniquement d'elle même ou des super classes de **l'arbre** d'héritage formé par les relations d'héritage simple. La structure concréte des objets est construite par héritage simple. Il n'y pas de réelle solution basée sur l'héritage multiple pour implémenter les fameux hotels restaurants en java.

Tout comme on parle de super-classes classes pour les classes héritées par une classe on parle de **super-interfaces**.

Les compilateurs vérifient qu'il n'y a pas de cycles dans le graphe d'héritage des interfaces.

6.5.2 Interfaces et masquage, ambiguïté

Une interface hérite de toutes les constantes définies dans ses super-interfaces. Les règles de masquages sur les variables d'une classe (cf. §4.3.5) s'appliquent aussi sur les constantes définies dans les interfaces.

Une classe implémentant une ou des interfaces héritent des définitions de constantes introduites par celles-ci et bien nentendu les règles s'appliquent aussi.

Le mécanisme de masquage ne suffit pas pour éviter des situations de **notations ambigues** d'accès aux constantes. Ceci ne limite absolument pas les constructions d'interfaces et classes par héritage et réalisation. Il peut juste se produire des erreurs de compilation lorsque l'on utilise la notation `<nom de constante>` pour accéder à une constante de même nom héritée simultanément de deux ou plusieurs interfaces ou classes. Pour lever l'ambiguïté il suffit d'utiliser la notation `<Nom de classe ou interface>.<nom de constante>`.

Nous allons donner un exemple volontairement artificiel pour rester simple et bien focaliser sur le problème. Considérons les deux interfaces et la classe de la figure 6.3 dont le diagramme de classe UML est donné sur la figure **??**. Les interfaces `I1` et `I2` introduisent toutes deux des constantes `A` et `B`, avec des expressions d'initialisations différentes (lignes 2, 3, 6 et 7). Ces déclarations définissent **quatres constantes différentes** que la classe `C` hérite puisse qu'elle indique implementer les interfaces `I1` et `I2` (ligne 9). La notation `B` (ligne 12) n'est pas ambigue : elle correspond à la déclaration de la ligne 10 qui masque celles héritées de `I1` et `I2`. Par contre la notation `A` de la ligne 14 est ambigue : le compilateur ne peut pas deviner si le programmeur désire faire référence à la déclaration héritée de `I1` ou celle héritée de `I2`. L'usage du nom de l'interface dans la référence (lignes 15 et 16) permet d'éviter un tel conflit. La figure 6.3 donne l'erreur générée par le compilateur dans une tel situation.

Figure 6.3 Interfaces, héritage mutiples et ambiguïté

```
interface I1 {
    int A=1;
    String B="trois";
}
interface I2 {
    int A=2;
    String B="deux";
}
class C implements I1, I2 {
    static double B=3.0;
    public static void main( String [] args ){
        System.out. println (B);
        System.out. print (A);
        System.out. println (I1.A);
        System.out. println (I2.A);
    }
}
```

6.5.3 Interfaces et surcharge

Une classe hérite de **toutes les déclarations** de méthodes d'instances faites dans les interfaces qu'elle (ou ses super-classes) prétend implémenter. Si plusieurs déclarations de méthodes correspondent au cas de surcharge d'une méthode d'instance (cf. §4.3.1) c'est à dire même nom et même liste de types pour les paramètres formels, toutes ces déclarations compte pour une, et ainsi une classe n'aura à fournir qu'une fois le code pour les déclarations de méthodes ainsi héritées. Dans ce cas toutes les déclarations et la définition effective doivent avoir le **même type de résultat** (cf. §4.3.5). Dans le cas contraire, le compilateur génère une erreur comme quoi les méthodes avec la même "signature" (nom et types des paramètres) doivent avoir le même *return type*.

6.5.4 Héritage multiple d'une même interface

Il est parfaitement possible qu'une interface hérite plusieurs fois d'une autre interface ou qu'une classe doive implementer plusieurs fois une même interface. Ces cas ne pose aucun problème : les déclarations de méthodes de l'interface vue plusieurs fois ont bien évidemment la même signature et sont donc regroupées deux à deux. Les constantes elles sont vue une seule fois : elles proviennent de la même interface avec les mêmes expressions d'initialisation, il n'y a donc aucune possibilité d'ambiguïté. Ce cas est proche de l'héritage "virtuel" en C++. On voit que finalement les restrictions faites sur les interfaces permettent un héritage multiple limité sans introduire de situtations subtiles.

6.6 Exemple dans la bibliothèque standard java

Les bibliothèques standards de java utilisent de nombreuses interfaces. Sinon les interfaces *listener* de l'AWT (*Advance Window Toolkit*). Ces *listeners* spécifient les actions que

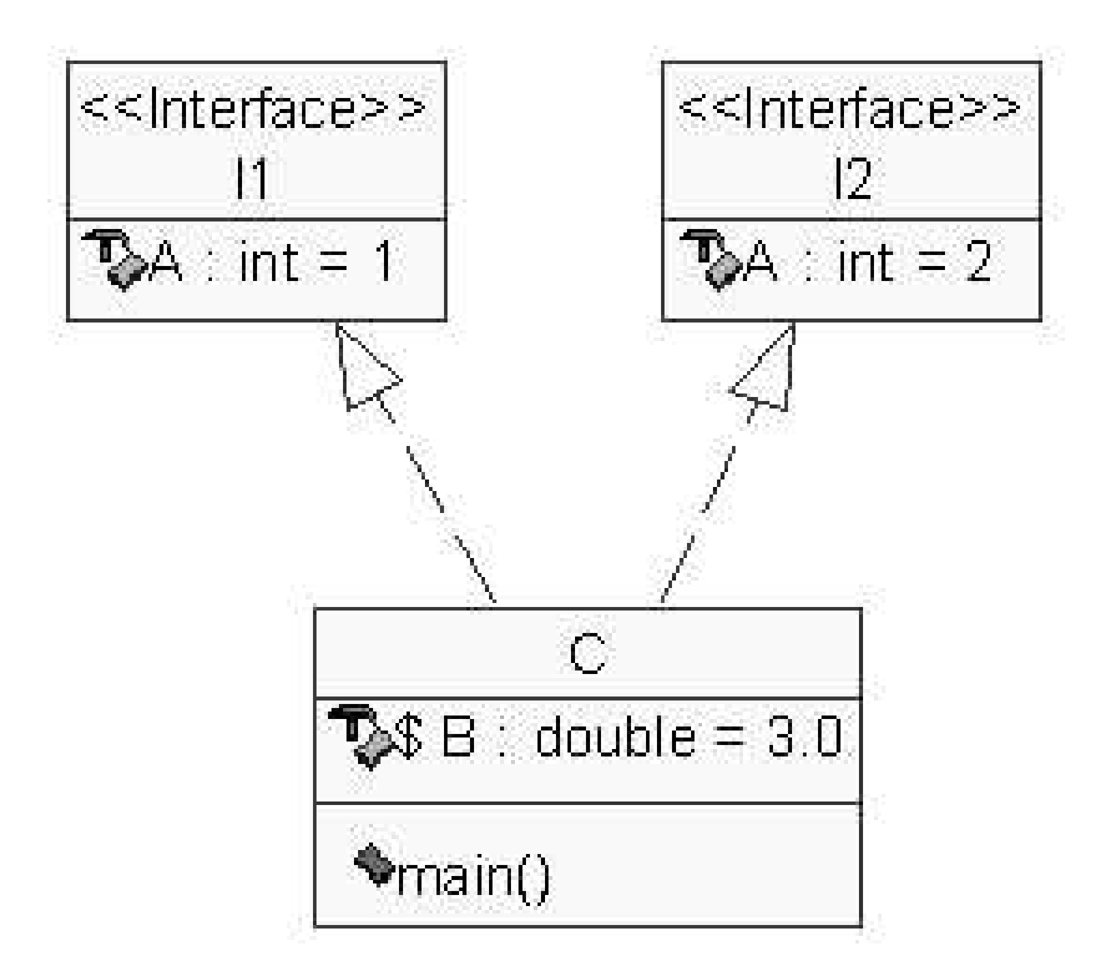


FIG. 6.2 – *Diagramme de classe et interfaces pouvant générer des ambiguïtés*

```
C.java :4: Reference to A is ambiguous. It is defined in interface I2 and interface I1.
        System.out. print (A);
                          ^
1 error
```

FIG. 6.3 – *Erreur de compilation en cas d'ambiguité sur des variables*

l'on peut faire en réponse à des évenements liés au divers objets d'une interface graphique : une fenêtre est fermée, un bouton est actionné. . .

Une interface "vide" : `java.util.EventListener` enracine l'ensemble des interfaces listeners.

Citons `java.awt.WindowListener` qui spécifie 7 méthodes. On y trouve la méthode `windowClosing` qui est appelée lorsque que l'utilisateur veut fermer la fenêtre via une interaction sur l'interface graphique.

Un programmeur désirant faire exécuter une action précise lorsque qu'un objet de la

Participant du *pattern*	classe java de l'exemple
Visitor	`JoueurVisitor`
ConcreteVisitor	`CalculMasseTotaleVisitor`
Element	`JoueurVisitorEnable`
ConcreteElement	`Joueur`, `Arme`, `Protection`, `Potion`, `Sac`, ...
ObjectStructure	`Joueur`

TAB. 6.1 – *Correspondance entre les composants du* pattern Visitor *l'aggrégat formé par la classe* `Joueur`

classe `Window` se ferme, doit créer d'une manière ou d'une autre une classe implantant l'interface `WindowListener`. Il doit ensuite créer une instance de cette classe et la liée à la fenêtre concernée en appelant la méthode d'instance `addWindowListener` de la classe `Window`.

6.7 Interfaces et *pattern visitor*

Le *pattern visitor* [**?**] est un cas de choix où une interface peut être définie et utilisée. Grossièrement ce *pattern* permet a un objet (d'une sous classe du participant `Visitor`) d'effectuer un calcul, de dérouler un algorithme sur une structure d'objets (participant `ObjectStructure`) **n'appartenant pas forcément à une même hiérarchie de classe**. Le *pattern* permet de définir plusieurs visiteurs (participant `ConcreteVisitor`) sous-classes de `Visitor` **sans modifier** les classes des objets visités. De plus le parcour s'effectue sans transtypage et selon les cas sans connaissance des relations entre objets. Dans la définition de [**?**] les classes des éléments visités (participant `ConcreteElement`) doivent tout de même toutes hériter d'une même classe abstraite (participant `Element`) qui définit une méthode `accept(Visitor)`.

Dans le cas de java qui ne possède pas d'héritage multiple on voit que l'utilisation d'une interface pour l'équivalent du participant `Element` s'impose. En général on réserve l'héritage pour définir ce que sont réellement les objets lors de l'analyse et les interfaces sont toutes indiquées pour introduire des concepts plutôt liés à la conception/implementation.

Sur notre exemple le joueur est une structure d'objets : lui-même et les équipements qu'ils portent à différents endroits (mains et dos). Le calcul à implanter à l'aide de ce *pattern* peut être la masse totale du joueur qui peut être utile si on introduit des règles concernant l'énergie consommée par le joueur. La correspondance entre les participants du *pattern* et les classes de notre exemple est donnée dans la table 6.1.

Nous n'avons pas tenu compte des autres sous classes de `Equipement` déjà introduites pour simplifier l'exemple.

La classe `JoueurVisitor` est définie comme une classe purement abstraite qui liste les méthodes de calcul à implanter pour chacune des *ConcreteElement*. Son code est donné dans la figure **??**. Nous avons fait le choix d'une classe plutôt que d'ue interface, car bien qu'une telle définition soit du domaine de la conception/implementation, il n'y a pas de raison pour qu'un objet appartenant à une autre hiérarchie doive aussi être un `JoueurVisitor`. C'est une classe purement technique qui doit être utilisée uniquement par dérivation.

Figure 6.4 Un visiteur abstrait pour la classe `Joueur`

```
public abstract class JoueurVisitor {
    public abstract void visiter ( Joueur j );
    public abstract void visiter ( Arme a );
    public abstract void visiter ( Protection p );
    public abstract void visiter ( Potion p );
    public abstract void visiter ( Sac s );
}
```

La figure **??** montre le code d'une implementation (par dérivation) de `JoueurVisitor`. Comme déjà dit elle calcule la masse totale d'un joueur et des équipements qu'il transporte. Pour cela elle définit une variable d'instance "d'accumulation" (ligne 2) qui est mise à jour chaque fois qu'un objet accepte d'être visité en appelant la méthode correspondant à sa classe concrête (lignes 4, 7, 10, 13, et 16). En générale la mise à jour ce fait en appellant la méthode `getMasse` (cf. §4.2), que nous avons d'ailleurs ajoutée à la classe `Joueur`. Nous considérons que la navigation : parcour des relations `porteEnMainDroite`, `porteEnMainGauche`, `porteSurDos`) ou traversée d'un conteneur comme le tableau de `Sac`, **est de la responsabilité des éléments eux-même**.

Le cas de la visite de la classe `Sac` pose un problème nous y avons en effet déjà défini une méthode `getMasse` qui prend en compte les équipements contenus dans le sac (cf. §4.4.1). Le visiteur passant individuellement sur chaque objet de la structure nous comptabiliserions deux fois les équipements contenus dans un sac si la méthode `getMasse` était utilisée lors de la visite d'un sac. Nous considérons donc qu'un sac possède une masse propre : sa masse à vide, retournée par la méthode `getMasseAvide` (ligne 16).

Figure 6.5 Un visiteur concrêt pour la classe `Joueur`

```
public class CalculMasseTotaleVisitor extends JoueurVisitor {
    private int masseTotale=0;
    public void visiter ( Joueur j ){
        masseTotale+=j.getMasse();
    }
    public void visiter ( Arme a ){
        masseTotale+=a.getMasse();
    }
    public void visiter ( Protection p ){
        masseTotale+=p.getMasse();
    }
    public void visiter ( Potion p ){
        masseTotale+=p.getMasse();
    }
    public void visiter ( Sac s ){
        masseTotale+=s.getMasseAvide();
    }
    public int getMasseTotale(){
        return masseTotale;
    }
}
```

L'interface `JoueurVisitable` donnée sur la figure 6.6, est très simple : elle ne spécifie qu'une seule méthode : `accept(JoueurVisitor)` qu'un visiteur doit appeler lorsqu'il veut visiter un objet. C'est le point principal de cet exemple. La figure 6.7 donne le une vue partielle du code de la classe `Joueur` implémentant l'interface `JoueurVisitable`. Nous avons dis qu'il est de la responsabilité des objets visités s'assurer la navigation. L'implémentation de la méthode `accept` de `Joueur` assure dont la propagation de la visite en appelant la méthode `accept` des équipements liés au joueur (lignes x, y, et z). Il faut noter que nous avons ici un problème au niveau du codage : un objet de la classe `Joueur` référence les objets aux extrémités des associations "porte..." grâce à des variables d'instances permettant d'accéder à des objets implementant l'interface `Portable` (`porteEnMainDroite`, `porteEnMainGauche`, ...) (cf. §6.4). Donc rien n'oblige *a priori* les objets *portable* d'implementer l'interface `JoueurVisitorEnable` et sa méthode `accept`, mais du point de vue de la visite du joueur on a envie que ces objets soient visités. Il faut donc que les les variables d'instances actuellement références vers l'interface `Portable`. Il faut donc modifier le typage de ces références. Plusieurs choix s'offrent à l'implementeur :

1. Faire hériter l'interface `Portable` de `JoueurVisitable`. Cette solution peut nuire à la réutilisation de `Portable` en effet elle oblige tout objet voulant implémenter l'interface `Portable` à fournir aussi la méthode `accept` de `JoueurVisitable`.
2. Créer une nouvelle interface *somme* de `Portable` et `JoueurVisitable`, typer les références à l'aide de celle-ci. Bien entendu il faudra que les équipements portables par le joueur et visitables à partir de celui-ci implémentent cette nouvelle interface.

La figure **??** illustre la deuxième possibilité : une interface `PortableEtJoueurVisitable` est définie par héritage multiple de `Portable` et `JoueurVisitable`.

Un problème similaire apparaît lors du codage de la méthode `accept` pour la classe `Sac` qui doit être aussi visitable. En effet il faut propager la visite au contenu du sac : un tableau de référence de la classe `Equipement`. Il faut donc que ces références nous permettent d'une manière ou d'une autre d'accéder à l'implémentation de l'interface `JoueurVisitable`. Les équipements sont *a priori* des objets dédiés à être ramassés par le joueur. Dans ce cas il ne semble par absurde d'imposer à **tout équipement** d'implémenter cet interface. Nous pouvons modifier la définition de la classe abstraite `Equipement` en déclarant qu'elle implémente l'interface `JoueurVisitable`. La figure 6.8 montre le code de cet ajout. C'est d'ailleurs le seul ajout fait dans la classe `Equipement`, la méthode `accept` requise par l'interface `JoueurVisitable` n'est pas implémentée. Elle reste donc une déclaration de méthode abstraite héritée par les sous classes de `Equipement`. Toutes les sous classes non abstraites de `Equipement` devront par contre implémenter concrêtement cette méthode (cf. §4.4). Du point de vu du typage, toute référence vers `Equipement` ou une de ses sous-classes est aussi une référence vers `JoueurVisitable`. La figure 6.10 montre comme la classe `Arme` peut implémenter les interfaces `Portable`, `JoueurVisitable` et `PortableEtJoueurVisitable`.

Figure 6.6 Une interface permettant à un objet d'être visité par un `JoueurVisitor`

```
interface JoueurVisitable {
    void accept( JoueurVisitor v );
}
```

Figure 6.7 Une classe `Joueur` pouvant être visitée par un `JoueurVisitor`

```
public class Joueur
implements JoueurVisitable {
    private int masse=80000;
    PortableEtJoueurVisitable porteEnMainDroite=null;
    PortableEtJoueurVisitable porteEnMainGauche=null;
    PortableEtJoueurVisitable porteSurLeDos=null;
    int getMasse(){
        return masse;
    }
    public void accept( JoueurVisitor v ){
        v.visiter(this);
        if (porteEnMainDroite!=null){
            porteEnMainDroite.accept(v);
        }
        if (porteEnMainGauche!=null){
            porteEnMainGauche.accept(v);
        }
        if (porteSurLeDos!=null){
            porteSurLeDos.accept(v);
        }
    }
}
```

Figure 6.8 Modification de la classe `Equipement` pour forcer l'interface `JoueurVisitor` dans ses sous-classes

```
public abstract class Equipement
implements JoueurVisitable {
    int masse;
    int valeur;

    int getMasse() {return masse;}
    void setMasse( int masse ) {this.masse=masse;}

    int getValeur() {return valeur;}
    void setValeur( int valeur ) {this.valeur=valeur;}
}
```

Figure 6.9 Une classe `Sac` pouvant être visitée par un `JoueurVisitor`

```
public class Sac extends Equipement
implements PortableEtJoueurVisitable {
    Joueur porteur;
    int emplacement=Portable.NON_PORTE;
    Equipement[] contient =new Equipement[10];
    int cptEquipement=0;
    int masseAvide=1500;
    void add(Equipement e){
        if (cptEquipement<contient.length){
            contient[cptEquipement]=e;
            cptEquipement++;
        } else {
            // exception ...
        }
    }
    void setValeur( int valeur ){}
    int getValeur(){
        int resultat =0;
        for( int i=0; i<cptEquipement; i++){
            resultat +=contient[i].getValeur();
        }
        return resultat;
    }
    void setMasse( int masse ){
        masseAvide=masse;
    }
    int getMasse(){
        int resultat =0;
        for( int i=0; i<cptEquipement; i++){
            resultat +=contient[i].getMasse();
        }
        return resultat;
    }
    // Ce que demande Portable.
    public Joueur getPorteur(){return this.porteur;}
    public int getEmplacement() {return this.emplacement;}
    public void porterMaintenant( Joueur j, int en ){
        this.porteur=j;
        this.emplacement=en;
    }

    int getMasseAvide() {
        return masseAvide;
    }
    public void accept( JoueurVisitor v ){
        for( int i=0; i<cptEquipement; i++){
            contient[i].accept(v);
        }
        v.visiter(this);
    }
}
```

Figure 6.10 Une classe `Arme` pouvant être visitée par un `JoueurVisitor`

```
public class Arme extends Equipement
implements PortableEtJoueurVisitable {
    private Joueur porteur;
    private int emplacement=Portable.NON_PORTE;
    int coefDegat;
    int getCoefDegat(){return coefDegat;}
    void setCoefDegat( int coef ){ this.coefDegat=coef;}
    public void accept( JoueurVisitor v){
        v.visiter(this);
    }
    // Ce que demande Portable.
    public Joueur getPorteur(){return this.porteur;}
    public int getEmplacement() {return this.emplacement;}
    public void porterMaintenant( Joueur j, int en ){
        this.porteur=j;
        this.emplacement=en;
    }
}
```

6.7.1 Remarques concernant cet exemple

1. Dans la spécification de l'interface de `JoueurVisitor` nous avons utilisé la **surchage paramétrique** pour différencier les différentes méthodes visitant les différents *concreteElements*. On retrouve en effet les différentes classes concrêtes à visiter dans le type du paramètres des méthodes `visiter`. Une autre possibilité serait d'utiliser des **noms de méthodes** différents pour chaque cas (`visiterJoueur(Joueur)`, `visiterArme(Arme),...`). En procédant comme nous l'avons fait nous pouvons être tenter de définir un comportement par "défaut" pour les équipements. En fait rien n'empêche de définir une méthode `accept(JoueurVisitor)` dans la classe équipement. La liaison dynamique toujours présente en java fera que cette méthode ne sera appelée sur un objet d'une sous-classe de `Equipement` que si aucune autre n'est définie dans l'arbre d'héritage entre `Equipement` et le type dynamique. Nous pouvons être tenter de définir une méthode `accept` dans la classe `Equipement` de la manière suivante :

```
public void accept( JoueurVisitor v ) {
        v.visiter(this);
}
```

Dans l'état actuel des choses ce code ne compile pas : il n'y a pas de méthode `visitor(Equipement)` dans la classe `JoueurVisitor` et le compilateur ne sélectionnera en aucun cas une sous-classes de `Equipement` comme type de paramètre formel pour correspondre à un paramètre effectif (type statique) de cette classe. Bref, aucune méthode `visiter` ne peut être sélectionnée. Il faudrait ajouter dans l'interface `JoueurVisitor` une méthode `visiter` avec un paramètre formel de type `Equipement`. Mais cette pratique va un peu à l'encontre de l'idée du *pattern* : il y

aura donc des classes concrêtes avec une méthode `visiter` et d'autres donc la visite est prise en compte dans une méthode plus générale. Ceci peut être confus pour le programmeur. De plus comme nous l'avons dans le chapitre sur la liaison dynamique ce genre de sucharge paramétrique peut générer des exécutions “peu naturelles”, sources d'erreurs de programmation.

2. Le lecteur peut constater que les classes `Arme` et `Sac` héritent deux fois de l'interface `JoueurVisitable`. Ceci est en quelque sorte involontaire, c'est une simple effet des modifications faites pour la réalisation physique. De toute façon n'a aucune influence génante (cf. §6.5.4).

6.8 Questions/remarques

Une classe qui déclare implementer une interface (mot clef **implements**) mais qui ne définit pas toutes les méthodes requises par cette dernière doit être déclarée comme abstraite (cf. §4.4) : les méthodes manquantes pour l'implementation complète de l'interface devront être introduites dans les sous classes. Le compilateur vérifie tout cela et génèrent les erreurs associées. Nous avons évoqué cet aspect sur la classe `Equipement` (cf. §6.7).

Classes et interfaces partagent le même espace de nomage, une classe et une interface ne peuvent donc pas avoir le même nom.

7

Les packages

7.1 Idée de base

Jusqu'à présent nous avons nommé nos classes (`Equipement`, `Joueur`, ...) ou interfaces (`Portable`, ...) sans trop nous poser de questions. Le nom indiqué dans la clause `class` nous a permis de les utiliser dans le reste de notre code.

Ce nomage "plat" n'est qu'une utilisation réduite de ce que propose java, il est possible de regrouper des définitions de classes et d'interfaces dans un même *package*.

Un *package* permet de définir des bibliothéques de codes et ainsi facilite la :

– réutilisation
– distribution
– structuration d'une application pour éviter des conflits de noms entre parties développées par des équipes différentes.
– ...

L'effet principal des *package* est de structurer l'espace de nom des classes et interfaces. La notion de package va aussi introduire quelques raffinements aux contrôles d'accès (classes et éléments d'une classe) vue jusqu'ici.

7.2 Création, utilisation, noms

Syntaxiquement, un package ne se crée pas, il existe implicitement lorsque que l'on place une classe dans un package grâce à la clause **package**.

Un nom de package peut se structurer comme un nom de répertoire, mais en utilisant des points (`.`) pour séparer un package d'un sous package. Si un package, voir une application, a pour vocation d'être distribuée via Internet, il est nécessaire de définir des noms de packages **uniques**. La spécification du langage Java recommande d'utiliser le nom de domaine Internet (garanti unique) de l'entreprise ou institution qui développe un package. Dans le cas du jeu "les tortues java" écrit par les auteurs tous membres du loria (domaine loria.fr) on peut choisir comme nom racine pour les packages du jeu : fr.loria.tortuesJava. Ainsi toute classe `C` définies dans le jeu et placées dans un sous package `sub` de `fr.loria.tortueJava` aura un nom **unique**, son *fully qualified name (FQN)* : `fr.loria.tortueJava.sub.C`. L'usage veut qu'un nom de package ou sous package commence par une minuscule et qu'un nom de classe ou interface commence par une majuscule. Les bibliothèques de base de java respectent ces conventions. Bien entendu, dans un même package les classes ou interfaces doivent avoir des parties droites de FQN (noms simples) différents.

Pour pouvoir utiliser les classes ou interfaces d'un autre package dans un fichier java ("unité de compilation"), afin de réaliser des sous classements, implementations d'interface, typages d'une variable ou d'un paramètre, instanciations, on peut utiliser le FQN d'une classe partout où nous avons utilisé un simple nom jusqu'à présent.

Si on fait un usage intensif de classes de plusieurs packages au nom un peu long (le package de l'interface graphique : java.awt par exemple), utiliser des FQN partout peut s'avérer très lourd. On peut alors utiliser en début de fichier, après l'éventuelle clause **package** une ou plusieurs clauses **import**.

Cette clause **import** a deux variantes :

1. **import** *<fully qualified name>*`;` pour pouvoir utiliser dans l'unité de compilation la classe unique désignée par la clause **juste par son, nom sans le nom du package**

2. **import** *<nom de package>*`.*;` pour pouvoir utiliser dans l'unité de compilation **toutes les classes** du package désigné directement par leur nom simple sans préfixe le nom du package. Attention, cette clause ne donnent **pas** accès aux classes des sous packages. De plus cette clause ne recherche une classe dans un package que lorsque qu'un "nom simple" est utilisé pour la première fois dans l'unité de compilit tion ("type à la demande").

7.2.1 Package par défaut

Jusqu'à présent nous n'avons pas utilisé de clause *package* on peut se demander alors comment nos classes sont utilisées et nommées. Elles sont en fait placées dans un package "non nommé". Il peut même selon le système de compilation employé avoir plusieurs packages non nommés. En tout état de cause il est fortement déconseillé de :

– D'utiliser les classes ou interfaces d'un package non nommé à partir d'un package nommé.

Il est fortement conseillé :

– De conditionner une application dans une hiérarchie de packages.
– De n'utiliser la facilité de package "non nommé" que pour des programmes de test.

Pour des packages largement distribuables nous avons vu qu'il faut utiliser le nom de domaine comme prefixe ce qui rend finalement cette discution sans intéret !

7.2.2 Quelques subtilités dans l'utilisation des clauses `import`

Le même nom simple de classe ou d'interface peut être utilisé dans différent packages. Ceci n'est bien entendu pas un problème puisque les packages ont des noms différents et donc les FQNs résultant seront bien différents. Mais, les clauses `import` permettent de désigner des classes ou interfaces avec leurs noms simples, il peut donc arriver qu'un nom simple puisse désigner plus d'une classe ou interface dans une unité de compilation.

Si le même nom simple est utilisé dans deux clauses `import` de la première forme concernant deux packages différents, une erreur de compilation est générée. En effet de telle clause charge immédiatement la classe ou interface concernée, il y a donc forcément ambiguïté si on utilise le nom simple dans l'unité de compilation. Les clauses ne servent donc à rien !

Par contre importer plusieurs fois un package par des clauses `import` de la deuxième forme n'est pas un problème. Un phénomène de masquage peut se produire car les clauses `import` de la première forme prime sur celles de la seconde. Si deux classes de même nom simple `A` sont placé dans deux packages, disons `p1` et `p2`, une unité de compilation utilisant les clauses :

```
import p1.A;
import p2.*;
```

dans son entête, accédera forcément à la classe `p1.A` par la notation `A`. Pour caccéder à `p2.A` il faudra forcément utiliser ce FQN.

Les situés dans le package par défaut masquent aussi celles importées par la clause de la deuxième forme.

Des masquages intempestifs peuvent se produire car le package `java.lang` est **importé par défaut** par une clause `import java.lang.*` dans toute unité de compilation. Il suffit de définir une classe de même nom simple qu'une classe prédéfinie de java (`Integer` par exemple), pour que cette dernière soit masquée dans le code.

7.3 Retour sur les contrôles d'accés, `public`, défaut

Maintenant que nous avons introduit plus précisemment la notion de package nous pouvons revenir sur les contrôles d'accès disponibles en java (cf. §4.6) car la notion d'appartenance d'une classe à un package est importante dans la définition de ceux-ci. De plus quand rien est précisé en java, l'accès est restreint aux classes d'un même package.

7.3.1 Accès à une classe ou un interface

Si la déclaration d'une classe (ou interface) est précédée du nom clef **public** elle peut-être utilisée dans n'importe qu'elle unité de compilation à condition que cette dernière possède une clause **import** appropriée, sauf si l'unité de compilation et la classe visée appartiennent au même package (il n'y a rien à préciser dans ce cas). Par utilisation il faut comprendre : typage de variables, paramètres et résultat de fonctions, utilisation comme super classe ou comme interface à implementer, création d'object à l'aide de l'opérateur **new**, et expressions de transtypage (*cast*).

Par contre, si le mot clef **public** ne précéde pas la déclaration d'une classe (ou interface), alors elle ne pourra être utilisée que par des unités de compilation **du même package**.

Il faut bien noter que se n'est pas parcequ'une classe n'est pas accessible à partir d'une unité de compilation, que cette dernière ne pourra pas manipuler d'instances dont le type dynamique est cette classe. Le contrôle d'accès ne lui permettra pas cependant d'y accéder avec des références du type statique de cette classe. Ce point sera illustré dans l'exemple du (cf. §**??**).

7.3.2 Accès aux éléments d'une classe

Sans utiliser les mots clefs **public**, **protected**, ou **private** lors de sa définition une variable ou méthode d'instance ou classe ne peut être accédée qu'à partir de code appartenant à des classes **du même package**.

L'introduction de la notion de package ne modifie en rien ce qui a été dit à propos de l'influence du mot clef **private** (cf. §4.6.1). Par contre nous devons compléter ce qui a été dit à propos du mot clef **protected** (cf. §4.6.2). En effet l'accès package "prime sur protected". En d'autre terme une classe peut utiliser des variables ou méthodes déclarées dans une autre comme **protected**, du moment qu'elle soit dans le même package, qu'elle soit sous classe de la deuxième **ou non**.

On peut donc conclure dire que le fait de grouper des classes dans un même package ouvre des possiblités de liens très forts ("fort couplage") entre elles. Par contre du point de

vue des classes situées dans d'autre package ce sont les éléments marqués comme **public** qui constituent des points d'accès dans le package.

7.4 Exemples

7.4.1 Une classe utilitaire

Nous allons créer une classe d'intêret général : un tableau d'objets qui s'aggrandit automatiquement, contrairement aux tableaux java qui ont une une taille fixée à leur création.

Cette classe peut servir à l'implementation d'un certain nombre d'autres. Nous allons la placer dans un package de classes utilitaires : `fr.loria.tortueJava.util`. Le code de cette classe est donné sur la figure 7.1. La ligne importante est donc la clause `package fr.loria.tortueJava.util`. Il faut aussi noter que nous avons déclaré cette classe comme publiques ainsi que tous les constructeurs et les méthodes comme publiques, car en effet sans indication particulière ces appels ne seraient accessibles qu'à partir des classes de ce package `fr.loria.tortueJava.util` ce qui est bien trop restrictif pour une classe "utilitaire". Une telle doit pouvoir être instanciée ou dérivée à partir d'une classe située dans n'importe quel package. De même, toute instance de cette classe doit pouvoir être créée, modifiée, accédée à partir de n'importe quel package. Un exemple d'utilisation de cette classe est donnée sur la figure 7.2 où on voit l'utilisation d'une clause `import` permettant d'utiliser la classe `TabDyn` avec juste la dernière partie de son FQN.

Figure 7.1 Exemple de classe placée dans un package

```
package fr.loria.tortueJava.util;

public class TabDyn {
    public TabDyn(int tailleInitiale, int tailleIncrement){
        this.elements=new Object[tailleInitiale];
        this.tailleIncrement = tailleIncrement;
    }
    public TabDyn(int tailleInitiale){
        this(tailleInitiale, 10);
    }
    public TabDyn(){this(10, 10);}
    public Object get(int idx){
        return elements[idx];
    }
    public void set(int idx, Object obj){
        if (idx>=length()){
            Object[] ancienElements=this.elements;
            elements=new Object[((idx/tailleIncrement)+1)*tailleIncrement];
            System.arraycopy(ancienElements, 0, elements, 0, ancienElements.length);
        }
        elements[idx]=obj;
    }
    public int length(){
        return elements.length;
    }
    private Object[] elements;
    private int tailleIncrement;
}
```

Figure 7.2 Exemple d'utilisation d'une classe placée dans un package

```
import fr.loria.tortueJava.util.TabDyn;

class TestTabDyn {
    public static void main(String[] args){
        TabDyn t=new TabDyn(1,1);
        for (int i=0; i<10000; i++){
            t.set(i, new Integer(i));
            System.out.println(t.get(i));
        }
    }
}
```

7.4.2 Groupements de classes et *factory*

Nous allons traiter un deuxième exemple plus sophistiqué qui est met en œuvre le *pattern abstract factory* [**?**].

Des monstres, les “Maulhy” ont été précédemment introduits (cf. §3.2). Comme indiqué dans le “cahier des charges des tortuesjava” (cf. §2) d’autres formes de monstres existent : les “Handrey” et les “Mhâziny”.

Du point de vue du jeu, pour sa logique, seule une classe partiellement abstraite “Monstre” est nécessaire. Il y sera précisé les capacités que doit présenter un monstre pour que le moteur du jeu puisse la manipuler : `combattre`, `setVie`, `deplacer`, `...`

Par contre, on peut imaginer plein de classes concrètes de monstre selon leur rendu graphique, les tactiques de combats et autres comportements . . .. Le jeu n’a pas connaitre exactement les classes concrêtes. Il peut se contenter de connaitre la classe générale des monstres, et une autres classes générale (ou interface abstraite) permettant de fabriquer des monstres concrêts.

Nous pouvons donc introduire une classe abstraite `Monstre` et une interface `FabriqueDeMonstre` dans le package `fr.loria.tortueJava.moteur` où les classes nécessaires à la logique du jeu seront placées. Par contre, les classes de monstres concrêts et la fabrique associée seront groupés dans un package à part, disons `fr.loria.tortueJava.config1`. On peut imaginer que d’autres configurations de monstres puisse être produites dans d’autres packages et insérées dans le jeu par des mécanisme non décrits ici (délégation vers les fabriques).

Le shéma UML des classes et interfaces mises en jeux est donné sur la figure **??**. Du point de vue du *patter abstract factory* la table **??** donne la correspondance entre participant du *pattern* et classes ou interfaces de l’exemple.

La figure 7.3 donne le code de toutes les classes ou interfaces java mises en cause dans l’exemple.

Du point de vue des packages et des contrôles d’accès, il faut noter que :

– L’interface `FabriqueDeMonstres` est déclarée comme **public**, car en effet l’idée est bien qu’une fabrique concrète de monstre, c’est à dire une classe implémentant cette interface puisque être localisée dans n’importe quel package voulant fournir un ensemble de monstre.
– Il en est de même pour la classe abstraite `Monstre` qui est faite (entre autre) pour être sous-classée dans d’autres packages.
– Le package `fr.loria.tortueJava.config1` est un exemple de tels packages, il contient :
  – une classe publique, `LeLoria` implémentant `FabriqueDeMonstre`. Cette classe possède un constructeur publique accessible de l’extérieur.
  – des sous classes de `Monstre` uniquement accessible par les autres classes de ce package.

Du point de vue de l’implémentation de la fabrique concrète, nous choisit la solution la plus simple, avec juste un petit artifice : le choix de la classe du monstre à créer est tiré au hasard (ligne 57de (cf. Fig. 7.3)). Un autre raffinement serait d’assurer qu’une seule instance de `LeLoria` ne soit créé dans une machine virtuelle donnée. Il faudrait utiliser le *pattern singleton* pour cette fabrique (cf. §3.9).

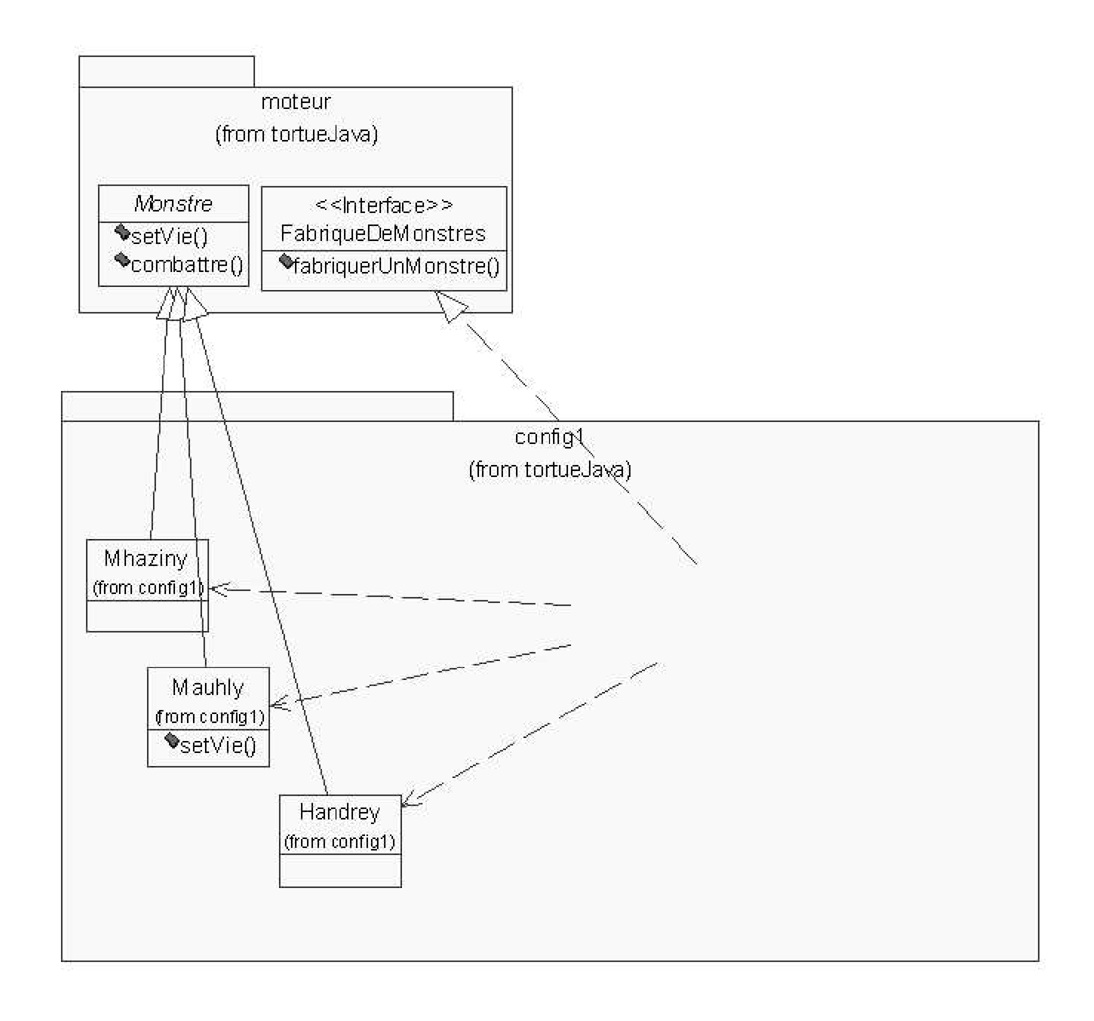


FIG. 7.1 – *Diagramme de classe d'un exemple de* pattern abstract factory

Participant du *pattern*	classe java de l'exemple
AbstractProductA	`Monstre`
ProductA1	`Mhaziny`
ProductA2	`Mauhly`
ProductA3	`Handrey`
AbstractFactory	`FabriqueDeMonstres`
ConcreteFactory1	`LeLoria`

TAB. 7.1 – *Correspondance entre les composants du* pattern abstract factory *et les classes liées aux monstres*

Figure 7.3 Monstres et fabriques abstraites

```
package fr.loria.tortueJava.moteur;
public abstract class Monstre {
    protected Monstre(int viep){
        vie=viep;
    }
    public void setVie(int viep){
        this.vie=viep;
    }
    public abstract void combattre(Monstre ennemi);
    private int vie;
}
package fr.loria.tortueJava.moteur;
public interface FabriqueDeMonstres {
    Monstre fabriquerUnMonstre();
}
package fr.loria.tortueJava.config1;
import fr.loria.tortueJava.moteur.Monstre;
class Mhaziny extends Monstre {
    Mhaziny(){
        super(10);
    }
    public void combattre(Monstre ennemi){
        ennemi.setVie(0);
    }
}
package fr.loria.tortueJava.config1;
import fr.loria.tortueJava.moteur.Monstre;
class Mauhly extends Monstre {
    Mauhly(){
        super(15);
    }
    public void combattre(Monstre ennemi){
        if (!(ennemi instanceof Mhaziny)) {
            ennemi.setVie(0);
        } else {
            this.setVie(0);
        }
    }
}
package fr.loria.tortueJava.config1;
import fr.loria.tortueJava.moteur.Monstre;
class Handrey extends Monstre {
    Handrey(){
        super(20);
    }
    public void combattre(Monstre ennemi){
        this.setVie(0);
    }
}

package fr.loria.tortueJava.config1;
import java.util.Random;
import fr.loria.tortueJava.moteur.Monstre;
import fr.loria.tortueJava.moteur.FabriqueDeMonstres;
public class LeLoria implements FabriqueDeMonstres {
    public Monstre fabriquerUnMonstre(){
        switch (rand.nextInt(6)){
        case 0: return new Mhaziny();
        case 1: case 2: return new Mauhly();
```

Il nous reste à esquisser l'utilisation de notre fabrique par un code client. La solution la plus simple est donnée sur la figure 7.4. Dans cet exemple (placé dans le package sans nom) : nous créons (ligne 6) une instance de `LeLoria` (publique avec un contructeur publique) manipulée par `fab` une référence d'interface `FabriqueDeMonstres` (interface publique). Puis grâce à cette instance nous créons des monstres par la méthode publique `fabriquerUnMonstre` manipulés via une référence à un `Monstre`. Les différentes clauses `import` simplifient l'écriture. Nous pouvons parfaitement déclarer pour cet exemple directement `fab` comme une référence d'objet `LeLoria` au lieu d'une référence d'objet implémentant l'interface `FabriqueDeMonstres`.

Figure 7.4 Utilisation simple d'une fabrique

```
import fr.loria.tortueJava.moteur.Monstre;
import fr.loria.tortueJava.moteur.FabriqueDeMonstres;
import fr.loria.tortueJava.config1.LeLoria;

class TestFabrique1 {
    public static void main(String[] args) {
        FabriqueDeMonstres fab= new LeLoria();
        for (int i=0; i<10; i++) {
            System.out.println(fab.fabriquerUnMonstre());

        }
    }
}
```

Modifions cet exemple pour que le lien vers la fabrique concrète ne soit plus fait directement dans le code, et donc par le compilateur, mais plutôt à l'exécution. Ceci est fait sur l'exemple de la figure **??**. Le mot (*FQN*)de la classe la fabrique concrète de monstres est passée en paramètre à l'exécution du programme :

```
java TestFabrique2 fr.loria.tortueJava.config1.LeLoria
```

Ce paramètre est récupéré dans le programme via la chaîne de caractère `args[0]` (cf. §3.3). La méthode statique `forName` de la classe `Class` du package prédéfini `java.lang` permet de charger dynamiquement une classe à partir de son FQN précisé sous forme d'une chaîne de caractères. Le résultat de ce chargement est un objet de la classe `Class` qui représente la classe désirée à l'exécution. Cette instance de la classe `Class` construit des instances de la classe quelle représente lorsqu'on invoque sa méthode `newInstance`. Nous avons enchaîné le chargement de la classe passée en paramètre et la création d'une de ses instances dans une seule expression (ligne 6). La méthode `newInstance` a un résultat typer en `Object` car en effet c'est l'unique classe qui puisse contenir n'importe quelle instance de n'importe quelle classe. Dans notre cas, nous pouvons être plus précis : la classe considérée doit implémenter l'interface `FabriqueDeMonstre`, nous forçons donc, par transtypage (*cast*) l'affectation de cette instance dans la variable `fab` référencant des objets implémantant l'interface `FabriqueDeMonstre`. Si le paramètre donné à l'appel de notre programme n'est pas un FQN d'une classe accéssible ou qui n'implémente pas `FabriqueDeMonstre` un certain nombre d'exceptions peuvent survenir, telles `ClassNotFoundException`, ou `CastException`.

Une fois `fab` affectée par un objet correct, le code reste le même que dans l'exemple précédent. On voit donc qu'il est bien inutile de vouloir trop spécialiser le type de la référence `fab`. Une discussion similaire peut être tenue pour les références accédant aux monstres créés : les typées en `Monstre` suffit, et de toute façon dans le cas du chargement dynamique de la classe de la fabrique d'objet, il est impossible de connaitre *a priori* les classes concrètes des monstres.

Figure 7.5 Utilisation dynamique d'une fabrique

```
import fr.loria.tortueJava.moteur.Monstre;
import fr.loria.tortueJava.moteur.FabriqueDeMonstres;
import fr.loria.tortueJava.config1.LeLoria;

class TestFabrique1 {
    public static void main(String[] args) {
        FabriqueDeMonstres fab= new LeLoria();
        for (int i=0; i<10; i++) {
            System.out.println(fab.fabriquerUnMonstre());

        }
    }
}
```

7.5 Stockage des package, `CLASSPATH` et autres détails

On peut se demander quelle influence ont les packages sur le rangement des classes dans un système d'exploitation. Cette question n'est en fait pas du ressort de la spécification du langage : chaque système de développement java est libre d'organiser physiquement les choses comme il l'entend. En fait, une machine virtuelle java, charge une classe en donnant un FQN à un objet chargeur de de classe (`ClassLoader`. Une application peut même définir des chargeurs de classes spécifiques, pour récupérer le code de certaines classes sur un serveur de base de donnée distant.

Mais, dans le cas général le stockage des classes (outils du jdk sun entre autres) les classes sont stockées dans :

1. Des arborescences de répertoires du système de fichiers du système d'exploitation supportant la machine virtuelle. Cette arborescence reproduit les imbriquations des noms de packages.
2. Des fichiers compressés d'archives avec une struture interne arborescente reproduisant aussi celles des noms de packages. Ce sont les fichiers d'archives Java : *Java ARchives files* ou *jar file*. Ce format simplifie la distribution des packages à travers un réseau, sur des CR .... Ils peuvent de plus contenir des informations sur les versions des packages, des signatures électriques pour être sûr que de l'organisation ou entreprise qui prêtant avoir développement les classes.

Dans le cas du jdk sun, l'emplacement les packages de base de java (`java.lang`, `java.UTIL?...`) sont connus directement par les outils (`javac`, `java`).

Par contre si vous installez d'autres packages ou que vous en développez il faut indiquer la liste des chemins d'accès aux répertoires contenant les racines des arborescences de fichiers supportant les packages dans la variable d'environnement `CLASSPATH`. En général si la variable `CLASSPATH` est utilisée il faut y placer le chemin . sur le répertoire courant, en effet c'est là que sont placées les classes du package non nomé. Pour les packages contenus dans des jarfiles, il faut placer le nom complet (avec chemin d'accès) du fichier d'archive. La syntaxe exacte des chemins d'accès et les séparateurs utilisés (; ou :) dépendent du système d'exploitation support.

Les outils du jdk acceptent aussi une option en ligne : `-classpath` pour indiquer une liste de chemins similaire à celle contenue dans la variable `CLASSPATH`. Si cette option est utilisée le contenu de la variable `CLASSPATH` n'est pas pris en compte.

Le compilateur, `javac` demande que les fichiers sources des classes et interfaces soient rangés dans **une arborescence de répertoires reflétant la structures des packages**. Une autre option (`-d` permet de générer les fichiers .class dans une arborescence similaire mais séparées des sources. Nous avons incitons à consulter la documentation html des outils du jdk, et de toute façon attendez-vous à quelques oublis et énervements concernant le "classpath" et ses amis !

Signalons aussi que le jdk fournie la commande `jar` pour créer des jarfiles. Cette commande a des options très proche de la commande `tar` d'Unix. Là aussi, les pages htlm de la documentation du jdk donnent toutes les précisions nécessaires , y compris sur les aspects formats de fichiers, signatures électroniques et manisfestes de distribution.

7.5.1 Application à l'exemple (cf. §7.4.1)

Dans le cas des exemples de la section précédente nous avons dans le répertoire courant l'arborescence suivante :

```
%ls -R .
TestFabrique1.java
TestFabrique2.java
TestTabDyn.java
fr:

loria
fr/loria:
tortueJava

fr/loria/tortueJava:
config1
moteur
util

fr/loria/tortueJava/config1:
Handrey.java
LeLoria.java
Mauhly.java
Mhaziny.java
```

```
fr/loria/tortueJava/moteur:
FabriqueDeMonstres.java
Monstre.java

fr/loria/tortueJava/util:
TabDyn.java
```

Pour compiler le deuxième exemple d'utilisation de la fabrique de monstres, et générer toutes les classes impliquées dans un répertoire temporaire (`/tmp`, nous pouvons taper la ligne de commande suivante :

```
javac -classpath . -d /tmp TestFabrique2.java
```

Après cette commande rien n'est apperu dans le répertoire courant, par contre dans `/tmp` on obtient :

```
TestFabrique2.class
fr

/tmp/fr:
loria

/tmp/fr/loria:
tortueJava

/tmp/fr/loria/tortueJava:
moteur

/tmp/fr/loria/tortueJava/moteur:
FabriqueDeMonstres.class
Monstre.class
```

On constate bien que le compilateur a reconstitué une aborescence similaire à celle de nos packages et des sources. On voit aussi que les classes du package sans nom sont placés directement dans le répertoire précisé dans l'option `-d` (ici la classe `TestFabrique2`). Par contre le sous package `fr.loria.tortueJava.config1` n'a pas été généré ! En effet la classe `TestFabrique2` ou les classes qu'elle utilse ne référencent aucune classe de `config1`, puisque nous avons volontairement utilisé un mécanisme de chargement dynamique à l'exécution. Pour la générer la classe `leLoria`, nous pouvons lancer, toujours à partir du même répertoire par une ligne de commande de la forme :

```
javac -classpath . -d /tmp fr/loria/tortueJava/config1/LeLoria
```

La classe `LeLoria` référence explicitement les autres classes du package `config1`, elles sont donc aussi générérées, dans un nouveau répertoire `config1`, au bin endroit dans `/tmp`.

Pour exécuter le programme ainsi généré (la procédure `main` est dans la classe `TestFabrique2`), on peut utiliser la commande en ligne :

```
java -classpath /tmp TestFabrique2 fr.loria.tortueJava.config1.LeLoria
```

8
Exceptions et entrées-sorties

8.1 Idée de base

Java propose un mécanisme d'exception, pour traiter les cas d'erreurs dûs :

– à la mauvaise utilisation de la machine virtuelle, ou des classes prédéfinies : classes introuvables, accès incorrect dans un tableau, division par zéro, ...
– à des cas anormaux à l'exécution : plus de mémoire disponible, canal de communication fermé de manière impromptue, ...

Ce mécanisme est extensible au code de l'utilisateur : le programmeur peut définir ses propres exceptions pour que les utilisateurs de ses classes et packages puissent gérer les erreurs d'une manière uniforme.

Lorsque qu'une exception surgit le flot normal des instructions du programme n'est plus suivi. La machine virtuelle cherche dans la pile à l'exécution un bout de code, installé par le programmeur, succeptible de traiter le problème.

Les motivations principales du choix d'un système d'exception très complet dans java [**?**][page 201], [**?**][Lesson: Handling Errors with Exceptions] sont :

– Forcer le programmeur à prendre compte, même de manière simple les erreurs se produisant dans son code. En effet c'est un discours classique chez les étudiants : "on fait un programme qui marche, on traitera les erreurs après", alors que les dites erreurs ont pour effet de rendre très difficile la mise au point du "programme qui marche" ! Java présente donc l'interêt de couper court à toutes questions sur le sujet ! [1]
– Permettre au programmeur de séparer de manière flexible le code du cas correct, exécuté dans 90 % des cas, du traitement des erreurs. En effet, un code C sous Unix, est truffé de `if (.. == -1)` pour filtrer les retours d'appels systèmes en erreur : cela nuit à la lisibilté du code, on a parfois du mal à saisir ce que fait un extrait de code, il faut en effet recoller plein de morceaux de code dilués dans des expressions conditionnelles. Le tutorial java avance un taux d'augmentation du code de 400% dès que l'on ajoute le traitement des erreurs (cas d'un code faisant des entrées-sorties dans un fichier).

Un tel mécanisme est courant, il existe dans d'autres langages, C++ et Eiffel entres autres. L'idée existe aussi en quelque sorte dans les systèmes d'exploitation : "trap" lié à un coprocesseur arithmétique, certaines interruptions sous Unix. Dans Visual Basic l'instruction "*on error goto* " permet de traiter les erreurs venant du système et des bibliothèques prédéfinies. Les programmeurs C sous Unix expérimentés utilisent `setlongjump` et `longjump` pour mettre en place des mécanismes approchés.

Dans java le mécanisme est parfaitement intégré : par exemple il n'y a pas à se soucier des effets qu'un arrêt abrupt d'une méthode peut avoir sur le système de synchronisation. De plus il tire parfaitement avantage des aspects classes et objets du langage. On peut dire qu'en Java, la spécification des exceptions pouvant provenir d'une méthode complète la signature de celle-ci. Ceci est très vrai puisque le compilateur fait des vérifications statiques poussées concernant les exceptions.

1. Les auteurs sont des enseignants-chercheurs, qui cherchent à avoir la paix !

8.2 Anatomie et cycle de vie

Shématiquement une exception est une instance de classe particulière qui est confiée à l'environnement d'exécution afin de trouver un point dans la pile à l'exécution qui soit capable de la traiter, de la "capturer".

8.2.1 Des objets et des classes

D'apparition d'un problème se traduit par la création d'un objet et la communication de celui-ci au contrôle d'exécution de la machine virtuelle par l'instruction `throw <référence d'objet>`. Pour cette dernière action on parle souvent de "**lever** une exception". La classe de cet objet doit hériter impérativement de la classe prédéfinie `java.Throwable`. En général pour refléter des problèmes de logique dans l'utilisation d'une méthode on choisit de créer une instance d'une sous-classe de `java.Exception` et non sous-classe de `java.lang.RuntimeException`. Une telle exception est une exception pour laquelle la présence obligatoire d'un traitement adéquat sera vérifié par le compilateur (*checked exception*).

De le cas de notre jeu nous pouvons introduire une exception vérifiée : `DeplacementHorsDuPlateauException` qui sera créée et levée par les méthodes déplaçant des monstres ou des joueurs sur les cases du plateau de jeu. La classe `Exception` possède deux constructeurs : - l'un sans paramètre, - l'autre avec une chaîne de caractère : un message à associer à l'exception. Dans le cas notre exemple, présenté sur la figure 8.1 nous ne proposons qu'un seul constructeur avec 3 paramètres : la case à partir de laquelle un mouvement incorrect a été tenté, un entier codant la direction du mouvement fautif (nous y reviendrons par la suite), et une chaîne de caractères communiquée à la super-classe (appel à **super** ligne 4). Ceci illustre bien que les exceptions sont des classes comme les autres : on peut les spécialiser pour y ajouter des informations complémentaires afin de mieux décrire le problème rencontré. Nous avons mis un caractère souligné _ devant le nom de la variable d'instance `case` (ligne 10) car `case` est un mot clef réservé du langage, et cause donc de nombreuses erreurs de compilation si il est utilisé comme identificateur !

Figure 8.1 Exemple de classe d'exceptions

```
package fr.loria.tortueJava.moteur;
public class DeplacementHorsDuPlateauException extends TortueJavaException {
    public DeplacementHorsDuPlateauException(Case c, int direction, String s){
        super(s); //
        this._case=c;
        this.direction = direction;
    }
    public final Case getCase(){ return _case;}
    public final int getDirection(){ return direction;}
    private Case _case; //
    private int direction;
}
```

8.2.2 Lever d'une exception

Comme déjà dit, lorsqu'une méthode veut lever une exception elle en crée une instance, puis utilise l'instruction `throw`. Considérons l'extrait de code de la figure 8.2.

Figure 8.2 Création et lever d'une exception

```
private Case calculerCaseSuivante (Case depuis, int direction )
        throws DeplacementHorsDuPlateauException {
        int xsuivant=depuis.getX();
        int ysuivant=depuis.getY();
        switch( direction ){
        case NordOuest: xsuivant--;
             ysuivant--;
             break;
        case Nord: ysuivant--;
             break;
        //Les autres directions ...
        default: throw new Error("Plateau . caseSuivante␣:␣direction␣incorrecte" );
                 // {\'a}\label{ligne:exceptions:calculerCaseSuivant:error}{\'a}
        }
        if ((xsuivant <0)||(xsuivant>=cases.length )||
             (ysuivant <0)||(ysuivant>cases[0].length )) {
             throw new DeplacementHorsDuPlateauException(depuis, direction , "" );
                       // {\'a}\label{ligne:exception:calculerCaseSuivante:DeplacementHorsDuPlateau}{\'a}
        }
        return this.cases[xsuivant][ysuivant];
}
```

Sur la ligne **??** de cet exemple nous voyons une utilisation de la classe d'exception présentée dans l'exemple 8.1. En effet après mise à jour des coordonnées de la case cible du déplacement le test précédent cette ligne détecte quelles correspondent à une case extérieure au plateau. Nous créons donc une exception `DeplacementHorsDuPlateauException` sans message précis, que nous communiquons immédiatement à l'environnement d'exécution.

Il faut bien noter que cette utilisation de l'instruction **throw** provoque l'arrêt **immédiat** de la méthode `calculerCaseSuivante`. Cette exception est une exception vérifié donc avons donc indiquer lors de la déclaration de la méthode (ligne 1) que celle-ci est succeptible de la lever (utilisation de la clause **throws**). Une classe utilisatrice de cette méthode (par exemple l'interface permettant à l'utilisateur de déplacer le joueur) devra donc obligatoirement prendre en compte cette exception (dans le cas d'interface graphique une indication de déplacement incorrect serait délivrée à l'utilisateur).

Par contre la ligne **??** donne un exemple d'exception non vérifiée. Nous avons simplement levée une `Error` si le paramètre `direction` ne correspond à aucun des cas attendus (les constantes `Nord`, `NordOuest`), ... n'ont pas été présentées sur l'exemple. Cette erreur n'a pas être spécifiée dans la partie **throws** de la déclaration de la méthode. Nous aurions aussi pu lever une exception `java.lang.IllegalArgumentException` sous-classe de `java.lang.RuntimeException`.

Pourquoi avoir adopté une exception vérifiée dans un cas, et une non vérifiée dans un

autre ? Pour le premier cas (déplacement impossible) cela reflète une fonction désiré de la méthode : elle détermine la case suivante ou indique par une exception que le déplacement est impossible, les deux possibilités correspondent à un fonctionnement normal du code. Par contre l'autre cas, l'erreur, reflète une utilisation incorrecte de la méthode de la part du programmeur : il doit utiliser une des constantes prédéfinie. Si une telle erreur est levée le programmeur doit **corriger** son code pour qu'elle n'apparaîsse plus.

8.2.3 Propagation

Lorsqu'une exception vérifiée peut être levée dans le code d'une méthode elle doit être traitée (voir paragraphe suivant ou progagée vers le contexte (pile à l'exécution) du code qui a appelé cette méthode.

Pour qu'elle soit propagée il n'y a en fait rien à faire durant l'exécution : il suffit que l'exeption levée reste dans le contexte d'exécution. Il suffit donc juste que le compilateur accepte le fait que l'exeption n'est pas immédiatement prise en compte : il faut préciser que la méthode peut lever cette exception (ou une de ses super classes comme nous le verrons dans la suite **??**).

La figure **??** donne un exemple de propagation de l'exception levée dans la méthode de l'exemple 8.2 extrait de la classe `Plateau`. L'appel de la méthode `calculerCaseSuivante` (ligne 3) est, comme l'avons vu sur la ligne 12 de l'exemple précédent, succeptible de lever une exception. Si cela est le cas, cet invocation de méthode retourne immédiatement ainsi que celui de la méthode appelante : ici `deplacer`. L'exception levée `DeplacementHorsDuPlateauException` est une exception verifiée, donc cet arrêt et propagation n'est possible que grace à la clause **throws** de la ligne 2.

Figure 8.3 Propagation d'une exception

```
public void deplacer(Monstre qui, Case depuis, int direction)
  throws DeplacementHorsDuPlateauException {
        Case suivante = calculerCaseSuivante(depuis, direction);
        suivante.abrite(qui);
        depuis.abrite(null);
        qui.estSur(suivante);
}
```

Un certain nombre d'instructions du langage, ou de méthode des classes de base de l'environnement génèrent des exceptions non vérifiées. Citons-en deux exceptions :

– `java.lang.NullPointerException` qui est levée lorsque l'on essaye d'accéder à une variable ou une méthode à partir d'une référence d'objet **null**.
– `java.lang.ArrayIndexOutOfBoundsException` qui est levée lorsque l'on accède aux éléments d'un tableau avec un index négatif ou supérieur à la taille du tableau par la notation `[]` entre autres.

8.2.4 Capture

exemple

Après avoir vu quand et comment les exceptions sont être générées, voyons comment le programmeur peut les prendre en compte dans son code pour effecturer des traitements adaptés.

Pour capturer les exceptions qu'un bloc d'instructions peut lever il suffit de placer ce bloc dans une clause **try..catch**. Voyons cela sur l'exemple d'utilisation de la méthode `deplacer` de la figure 8.4.

Figure 8.4 Capture d'une exception

```
FabriqueDeMonstres fabMonstres =...
FabriqueDeCases fabCases =...
Plateau  p=new Plateau(fabCases,  16,  9);
Monstre m=fabMonstres.fabriquerUnMonstre();
p.poser(m, 0,  0);
Random rand=new Random();
for  (int  i=0;  i< 40;  i++){
    try  {
        p.deplacer(m, m.estSur (),   rand.nextInt (8)+1);
        System.out.println (p.toString  ());
    }catch  (DeplacementHorsDuPlateauException e){
        System.out.println ("La␣direction␣"+e.getDirection ()+
                            "␣n'est␣pas␣valide␣a␣partir␣de␣:␣"+
                            "x="+e.getCase ().getX()+
                            ",␣y="+e.getCase().getY ());
    }
}
```

syntaxe et fonctionnement

La syntaxe plus exacte de l'instruction **catch** est la suivante :

```
try  {
        ...  code  ...
} catch(Exception2 e1){
        ...  code  1...
} catch(Exception2 e2){
        ...  code  2...
}
...
finally {
        ...  code  ...
}
```

Explicitons plus en détail le fonctionnement à l'exécution de cette construction [**?**][§14.18].

Si une exception est levée pendant l'exécution du bloc suivant la clause **try** l'exécution du bloc est immédiatement arrêtée à l'endroit du code qui a succité cette exception.

La machine virtuelle essaie ensuite de transférer l'exécution à un des blocs de code suivant une clause **catch**. Pour sélectionner un bloc, l'environnement cherche **dans l'ordre de**

leur déclaration (ici `Exception1`, `Exception2`, `...`) le bloc de la clause **catch** où le type spécifié permet à la variable déclarée d'être affectée par une référence vers l'exception en cours de traitement. C'est bien entendu **le type dynamique** de celle-ci qui est considéré (cf. §3.3).

Une fois une clause **catch** sélectionnée l'exécution continue à partir du début du bloc lui correspondant jusqu'à la fin de celui-ci. Si le bloc est susceptible de lever des exceptions vérifiées elles doivent soit être capturées par une autre clause **try-catch** dans ce bloc, soit le bloc englobant la clause **try-catch** initiale devra capturer ou propager cette nouvelle exception.

Si une clause **finally** suit les clauses **try-catch** elle sera **toujours** exécutée après le bloc du **try** si il n'y a eu d'exception, ou le bloc de la clause **catch** sélectionnée dans le cas contraire.

Il faut noter qu'il est parfaitement possible d'écrire une clause **try** sans clauses **catch**, avec seulement une clause **finally**. Cette dernièr sera donc **toujours** exécutée.

8.3 Exception et héritage

Il est parfaitement possible qu'une exception levée dans un bloc d'une clause **finally** masque une exception levée dans un un bloc d'une clause **try** ou **catch**. Il en va de même pour la valeur du résultat d'une méthode positionné par **return** : il peut être masqué par une exception ou un autre **return** L'exemple **??** et la trace de son exécution **??**xhibe quelques cas de tels masquage. Le premier appel à méthode `m` correspond à une exécution sans surcharge. La clause **return** de la ligne 9 est vue normalement, puis le bloc de la clause **finally** est exécuté mais ne fait rien de particulier puisque le paramètre effectif vaut 1. Par contre, lors du deuxième appel la valeur positionné par la clause **return** de la ligne 9 est masquée pas celui de la ligne 13. Lors du troisième appel la valeur positionné est perdue car en fait le programme s'arrête brutalement lors de la création et du levé de l'exception de la ligne 14.

Figure 8.5 Exemple de classe d'exceptions

```
class Masquage {
    public static void main(String [] args) throws A{
        System.out. println ("m(1)="+m(1));
        System.out. println ("m(2)="+m(2));
        System.out. println ("m(3)="+m(3));
    }
    private static int m(int p) throws A {
        try{
            return 10;
        } finally {
            switch (p){
            case 1: break;
            case 2: return 20;
            default : throw new A();
            }
        }
    }
}
```

```
m(1)=10
m(2)=20
new A
        at Masquage.m(Masquage.java:14)
        at Masquage.main(Masquage.java:5)
Exception in thread "main"
```

FIG. 8.1 – *Exécution de l'exemple de la figure* **??**

9

Le graphique

9.1 Applications graphiques

Le programme principal d'une application graphique est le plus souvent construit sur le même cannevas:

1. Mise en place des écrans. Il s'agit de mettre en place des arbres d'objets graphiques. Les objets graphiques sont des objets à part entières qui incluent un état et la capacité d'être dessiné. Pour représenter un état de notre application tortues java, il faudra donc créer des objets graphiques et maintenir l'état des objets graphiques en accord avec l'état des objets applicatifs. Cette distinction objet graphiques/objets applicatifs est fondamantale et la manière de gérer les relations entre ces deux types d'objets est loin d'être triviale.
2. Mise en place des "callbacks". Il s'agit de mettre en place des règles du type : "On Event Do action" i.e. quand ce bouton est appuyé, alors exécuter cette méthode avec ces paramètres.
3. Lancement de la boucle d'événements. Un programme graphique passe le plus clair de son temps à attendre des événements généralements en provenance des objets graphiques. Quand un événement arrive, il faut examiner les desiderata du programmeur (ses callbacks) et exécuter la méthodes adéquates avec les paramètres adéquats.

Pour construire une interface graphique il faut donc être capable:

- de construire des arbres de composant graphiques.
- d'associer des actions aux événements.
- de gérer les relations objets applicatifs/objets graphiques. C'est à dire les objets graphiques doivent refléter l'état des objets applicatifs à tout moment.

9.2 Construire un écran graphique: le modèle composite

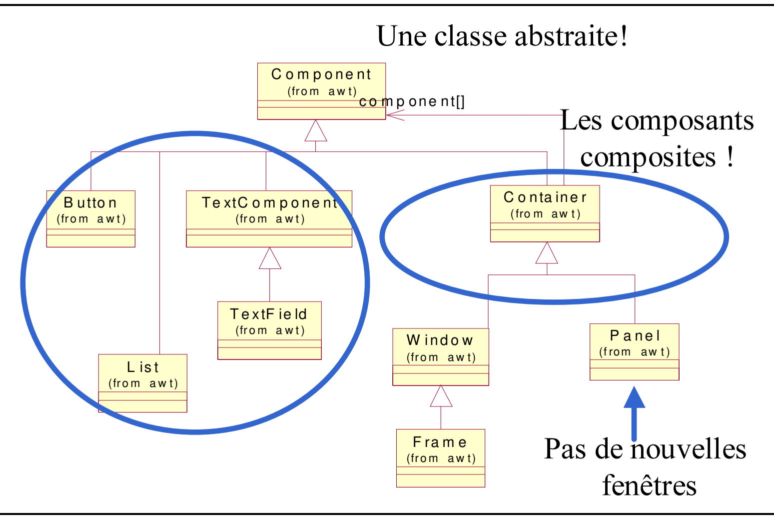
Nous avons déjà eu affaire au modèle composite avec nos tortues java. Le sac d'un monstre est un exemple de modèle composite. Il s'agit en fait d'un patron de conception (design pattern) bien connu.

Le modèle composite permet de créer des arbres en s'appuyant sur 3 types de classes:

- Les classes feuilles
- Les classes noeuds
- une classe abstraite permettant de manipuler indifférement une feuille ou un noeud.

9.2.1 Construction de l'arbre

Figure 9.1 Hiérarchie de classes principale



La figure 9.1 représente une partie du diagramme de classes de la librairie graphique java.

la classe abstraite. La classe abstraite $Component$ est la classe $Composant$ du modèle composite. Elle définit le comportement et l'état d'un objet graphique abstrait. De manière générale, L'état d'un objet graphique abstrait est définit par une position x,y et une taille. Le comportement essentiel d'un objet graphique est sa capacité a être dessiné à travers sa méthode $paint$.

Les feuilles. Les feuilles du modèle composite à savoir les classes $Button$,$List$,$Textfield$ . . . sont des objets graphiques primitifs prêt à l'emploi. Ils redéfinissent la méthode $paint$ en fonction de ce qu'ils sont, et emettent les événements en adéquation avec leurs rôles. Un $Button$ rédéfinit la méthode paint pour dessiner un bouton et peut émettre les événements "Bouton enfoncé" ou "bouton relaché".

les noeuds. Les noeuds servent à composer les composants. La classe $Containter$ est une classe abtraite permettant simplement d'aggréger des composants. Maintenabnt ces "répertoires" de composants peuvent s'afficher dans des fenêtres à part auquel cas, le noeud en question sera de type "Window" ou au sein d'une fenêtre existante auquel cas le noeuds sera de type "Panel".

Nous appliquons ce principe pour représenter l'état de notre heros de tortues java. Il s'agit de visualiser le nom, la vie, la force du heros, l'arme et l'armure portée et enfin le contenu du sac. La figure **??** représente les objets à visualiser. Nous appelerons ces objets le modèle. La figure 9.3 représente notre écran de visualisation. Il ságir dún assemblage d'objets graphiques représentant l' état du modèle.

Figure 9.2 Le modèle de l'application "sac"

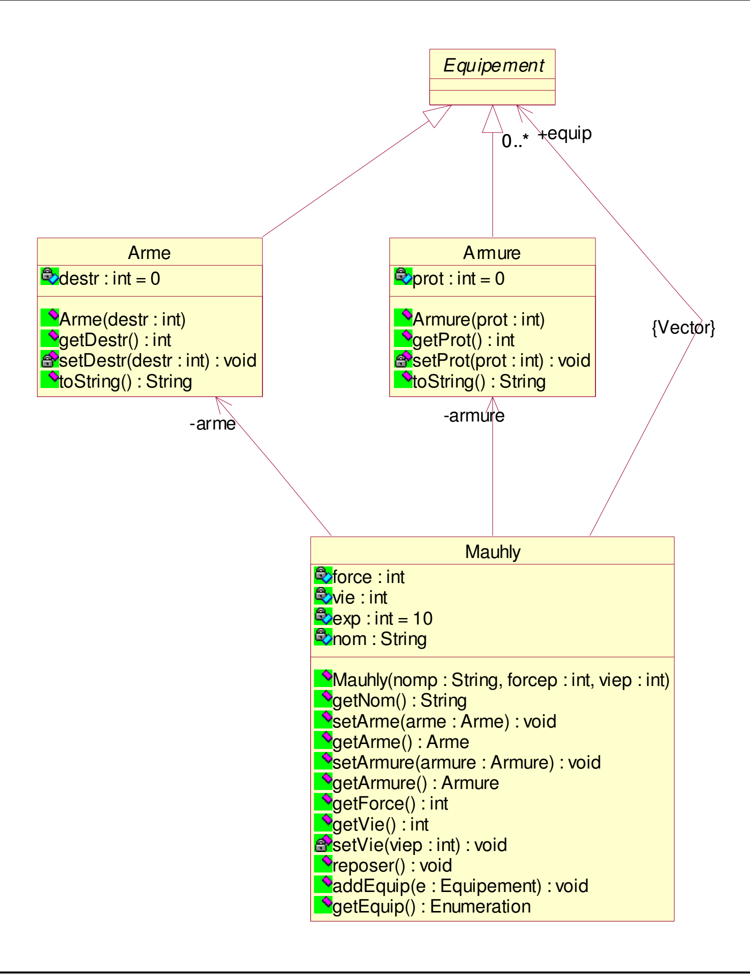
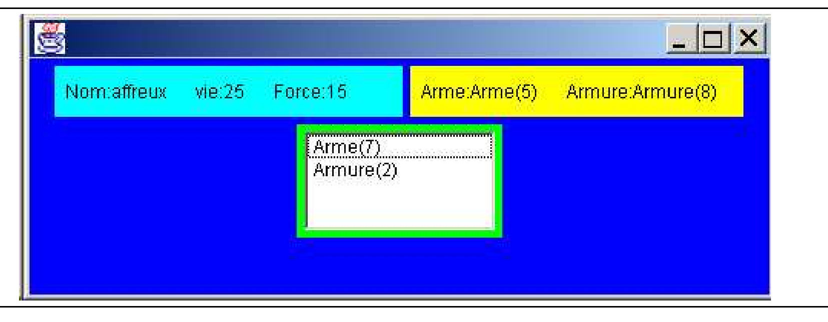
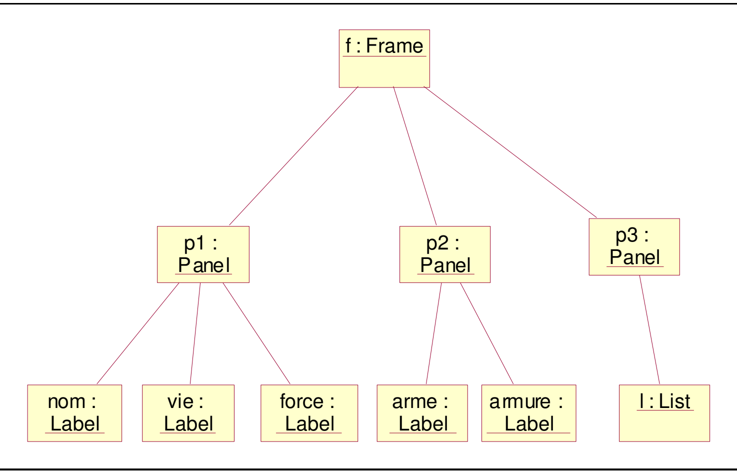


Figure 9.3 Le héro



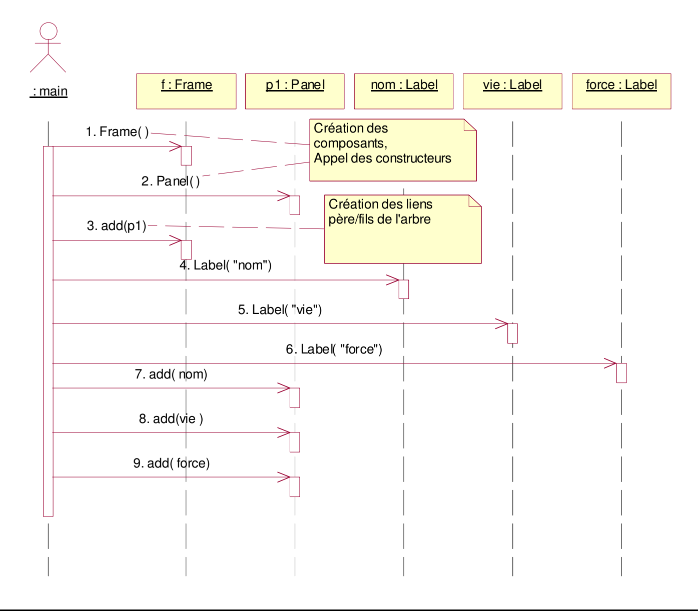
Nous avons symbolisé avec des couleurs différentes les différents noeuds de l'arbre de composants graphique. La figure 9.4 représente l'arbre des composants graphiques utilisés pour contruire l'écran de la figure 9.3.

Figure 9.4 Arbre de composants graphiques



La construction de l'arbre lui-même est triviale. Il faut créer les composants primitifs et les noeuds. Chaque noeuds dispose de méthodes “add(Component c)” conformément au modèle par composition. Il suffit donc d'enregistrer les composants dans les composites. La figure 9.5 donne le diagramme de séquence pour construire l'arbre.

Figure 9.5 Construction de l'arbre



Enfin, le code java est quasiment automatique.

```
import java. util .*;
import java .awt.*;

class Main {
    public static void main(String args []) {
        Mauhly m=new Mauhly("affreux",15,25); // mise en place du modèle
```

```
        m.setArme(new Arme(5));
        m.setArmure(new Armure(8));
        m.addEquip(new Arme(7));
        m.addEquip(new Armure(2));

        Frame f=new Frame(); // mise en place de la fenêtre principale
        f.setBackground(Color.blue);
        // ...

        // Noeud pour Nom, vie, force ..
        Panel p1=new Panel();
        p1.setBackground(Color.cyan);
        p1.add(new Label("Nom:"+m.getNom()));
        p1.add(new Label("vie:"+m.getVie()));
        p1.add(new Label("Force:"+m.getForce()));

        // Panel pour les armes et armure
        Panel p2= new Panel();
        p2.setBackground(Color.yellow);
        p2.add(new Label("Arme:"+m.getArme()));
        p2.add(new Label("Armure:"+m.getArmure()));

        // panel pour visualiser le contenu du sac
        Panel p3= new Panel();
        p3.setBackground(Color.green);
        java.awt.List l=new java.awt.List();

        p3.add(l);
        for (Enumeration e = m.getEquip() ; e.hasMoreElements() ;) {
            l.add(" "+e.nextElement());
        }

        // aggrégation des différents panel
        f.add(p1);
        f.add(p2);
        f.add(p3);

        // Arbre prêt : Affichage
        f.pack();
        f.setVisible(true);
    }
}
```

Pour construire cet écran, nous avons utilisé la librairie AWT. Nous pouvons construire la même interface, en suivant le même principe de composition mais avec une autre librairie: la librairie swing. Le programme est quasiment identique.

```
import java.util.*;
import javax.swing.*; // on utilise swing !
import java.awt.*;

class Main {
    public static void main(String args[]) {
        Mauhly m=new Mauhly("affreux",15,25);
```

```
        m.setArme(new Arme(5));
        m.setArmure(new Armure(8));
        m.addEquip(new Arme(7));
        m.addEquip(new Armure(2));

        JFrame f=new JFrame(); // attention ici c' est un JFrame
        f.getContentPane().setBackground(Color.blue);
        // ...

        JPanel p1=new JPanel(); // JPanel !
        p1.setBackground(Color.cyan);
        p1.add(new JLabel("Nom:"+m.getNom()));
        p1.add(new JLabel("vie:"+m.getVie()));
        p1.add(new JLabel("Force:"+m.getForce()));

        JPanel p2= new JPanel();
        p2.setBackground(Color.yellow);
        p2.add(new JLabel("Arme:"+m.getArme()));
        p2.add(new JLabel("Armure"+m.getArmure()));


        JPanel p3= new JPanel();
        p3.setBackground(Color.green);
        JList l=new JList(m.getEquip());
        p3.add(l);

        // aggrégation des différents panel
        f.getContentPane().add(p1);
        f.getContentPane().add(p2);
        f.getContentPane().add(p3);

        // Arbre prêt : Affichage
        f.pack();
        f.setVisible(true);
    }
}
```

D'un point de vue conceptuel, les différences sont mineures. Sur cette application, les rendu graphique est quasiment identique. La différence entre les deux librairies est d'ordre architectural. La librairie AWT se base sur la librairie graphique du système hôte. Sur windows, si je crée un bouton java, la JVM va demander à la librairie graphique de windows de créer un bouton windows. La JVM se charge par la suite de gérer le mapping entre l'objet Java et l'objet graphique windows. Cette approche a l'avantage de déléguer l'affichage et le rendu des composants graphiques au système hôte. La même application va créer des boutons windows si elle est executée sur windows et des boutons MacOS si elle est executée sur MacOS.

La librairie "swing" ne délègue pas le rendu graphique des composants au système hôte. Elle gère elle-même comment une liste ou un label est dessiné. Elle demande juste au système hôte de lui fournir une fonction "drawpoint(x,y)". Une application graphique construite avec la librairie "swing" peut très bien sur une machine windows, dessiner ses boutons avec un "look-and-feel" MacOS.

La librairie AWT dépend pour beaucoup des possibilité de la librairie graphique du système hôte. Pour qu'un composant graphique soit présent dans la librairie AWT, il doit avoir un équivalent sur tous les systèmes hôtes avec le même comportement, ce qui n'est pas toujours le cas. La librairie swing s'affranchit de cette limitation et permet de mettre en place des composant graphique originaux et portables.

Construire l'arbre des composants est fondamental pour construire une interface graphique. Les feuilles fournissent les objets graphique primitifs, les noeuds se comportent comme des répertoires d'objets graphiques. Pour avoir un écran graphique bien formé, il faut en plus spécifier pour chacun des noeuds quelle politique utiliser pour placer les composants qu'il contient. C'est le rôle des algorithmes de placement que nous allons voir maintenant.

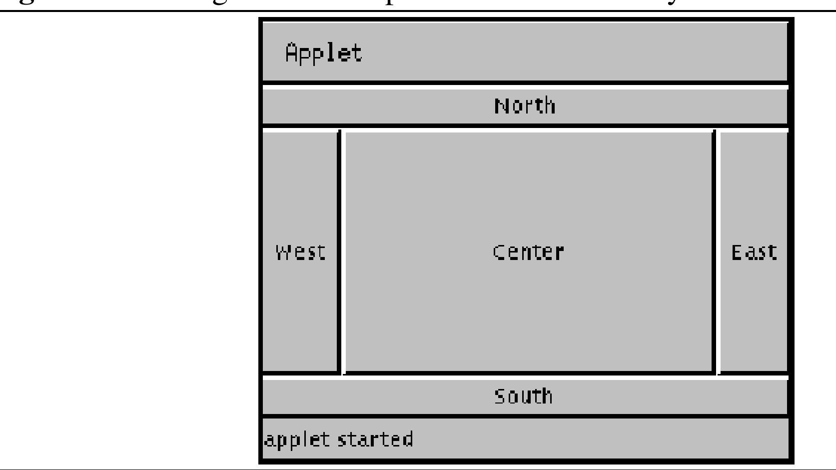
9.2.2 Gestion du placement

Chaque noeud d'un arbre graphique doit avoir une politique de placement des objets qu'il contient. Il faut se rappeler qu'un conteneur est aussi un composant et qu'à ce titre, il doit lui-aussi être dessiné. Comment va-t-il dessiner les objets qu'il contient ? à quelle position dans l'espace qui lui est propre ? avec quelle taille ?

Les algorithmes de placement sont là pour organiser les placements des objets graphiques d'un conteneur. Avoir des algorithmes de placement est plus interessant que de placer soit même les objets dans l'espace graphique du conteneur parce qu'ainsi, les redimensionnement de fenêtres ou les changement, l'apparation de nouveaux composants dans le conteneur vont provoquer à nouveau l'activation de l'algorithme de placement.

Les algorithmes de placement découpe l'espace graphique d'un conteneur en zones où ils placent les objets contenus dans le conteneurs. Si un conteneur contient 5 boutons et les place suivant un "BorderLayout", le résultat est celui de la figure 9.6.

Figure 9.6 Les algorithmes de placements: BorderLayout

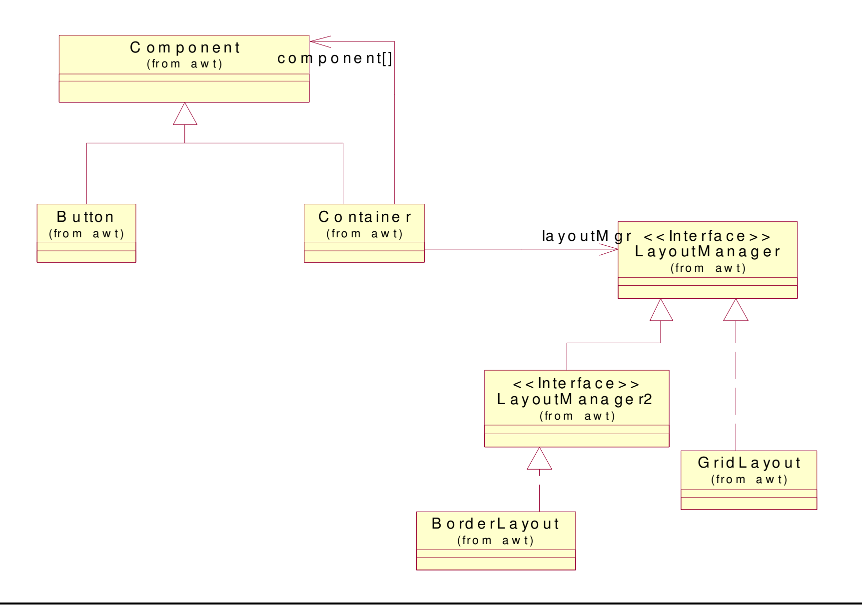


Si conteneur contient 6 boutons et utilise un algorithme de placement de type "GridLayout", le résultat est celui de la figure 9.7

Figure 9.7 Les algorithmes de placements: GridLayout



Figure 9.8 Les algorithmes de placements



Chaque conteneur va donc être associé à un algorithme de placement, algorithme qui est représenté sous forme d'un objet implémentant l'interface "LayoutManager". La figure 9.8 montre le diagramme de classe faisant apparaitre les relations entre les conteneurs, l'interface "LayoutManager" et les algorithmes de placement situé dans les classes BorderLayout, GridLayout...

Ce diagramme montre un bel exemple de délégation. Le conteneur délègue le placement des objets qu'il contient à une instance de LayoutManager. Si on inspecte la méthode doLayout() du conteneur, elle repose sur la définition de la méthode abstraite $layoutContainer()$ de l'interface LayoutManager.

```
public class Container extends Component {
 ...
    public void doLayout() {
        layout ();
    }
```

```
public void layout () {
    LayoutManager layoutMgr = this.layoutMgr;
    if (layoutMgr != null) {
        layoutMgr.layoutContainer(this); // delegation
    }
}
```

Figure 9.9 Instanciation des algorithmes de placement

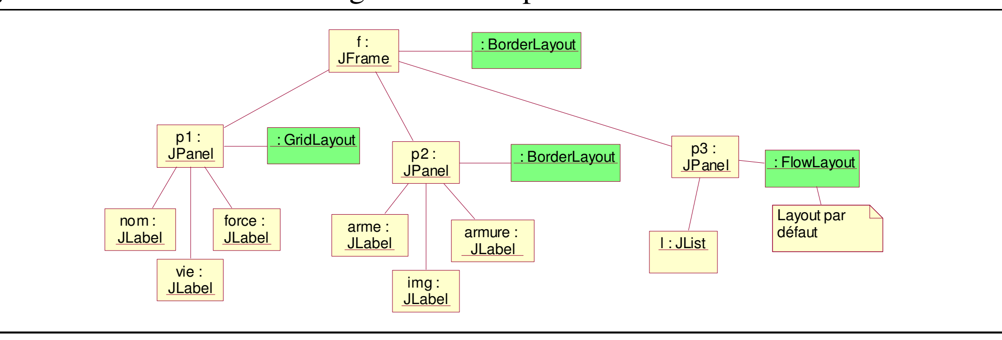
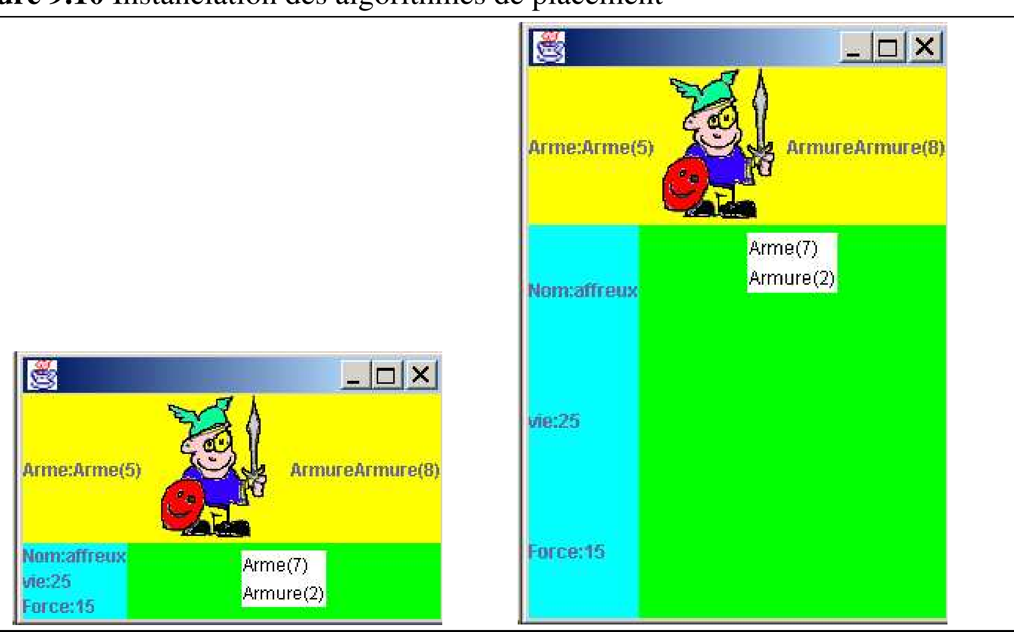


Figure 9.10 Instanciation des algorithmes de placement



Nous reprenons l'exemple de visualisation de l'état du héros. Chaque noeud de l'arbre de composant graphique va déléguer le gestion du placement à un algorithme de placement. La figure 9.9 représente quel algorithme de placement est associé à chaque noeud. La figure 9.10 représente le résultat à l'écran. Le code suivant permet de créer cet écran.

```
import java.util.*;
import javax.swing.*;
import java.awt.*;
```

```
class Main {
    public static void main(String args []) {
        Mauhly m=new Mauhly("affreux",15,25);
        m.setArme(new Arme(5));
        m.setArmure(new Armure(8));
        m.addEquip(new Arme(7));
        m.addEquip(new Armure(2));

        // Fenêtre principale
        JFrame f=new JFrame();
        f.getContentPane().setBackground(Color.blue);
        f.getContentPane().setLayout(new BorderLayout());

        // Noeud pour Nom, vie, force ..
        JPanel p1=new JPanel();
        p1.setLayout(new GridLayout(0,1));
        p1.setBackground(Color.cyan);
        p1.add(new JLabel("Nom:"+m.getNom()));
        p1.add(new JLabel("vie:"+m.getVie()));
        p1.add(new JLabel("Force:"+m.getForce()));

        // Panel pour les armes et armure
        JPanel p2= new JPanel();
        p2.setLayout(new BorderLayout());
        p2.setBackground(Color.yellow);
        p2.add(new JLabel("Arme:"+m.getArme()),BorderLayout.WEST);
        p2.add(new JLabel(new ImageIcon("heros.gif")), BorderLayout.CENTER);
        p2.add(new JLabel("Armure"+m.getArmure()),BorderLayout.EAST);

        // panel pour visualiser le contenu du sac
        JPanel p3= new JPanel();
        p3.setBackground(Color.green);
        JList l=new JList(m.getEquip());
        p3.add(l);

        // aggrégation des différents panel
        f.getContentPane().add(p1,BorderLayout.WEST);
        f.getContentPane().add(p2,BorderLayout.NORTH);
        f.getContentPane().add(p3,BorderLayout.CENTER);

        // Arbre prêt : Affichage
        f.pack();
        f.setVisible(true);
    }
}
```

Figure 9.11 Autre Instanciation des algorithmes de placement

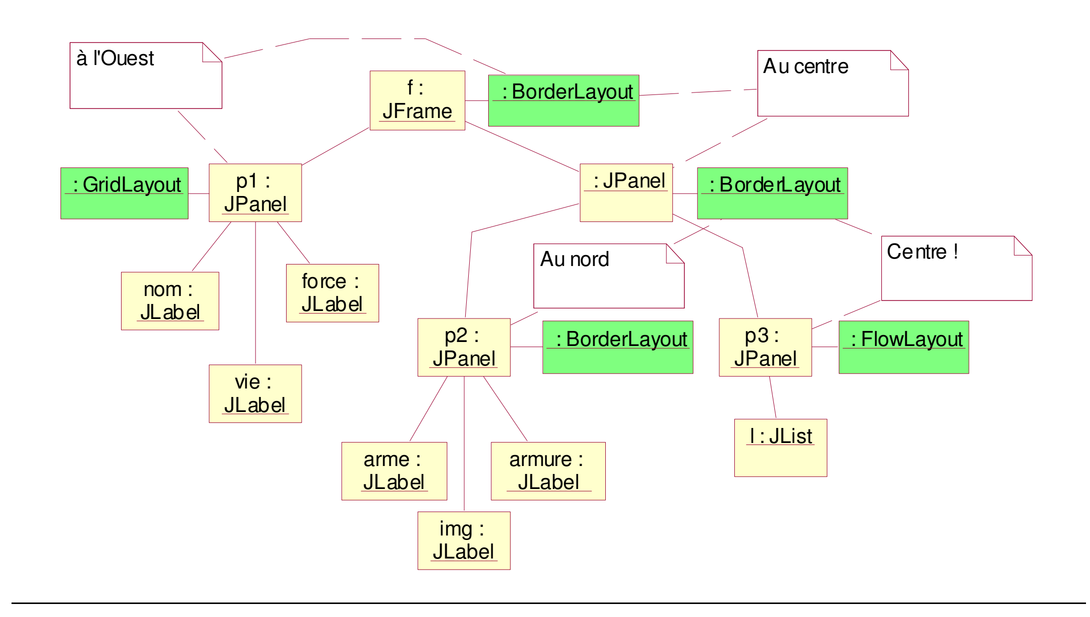
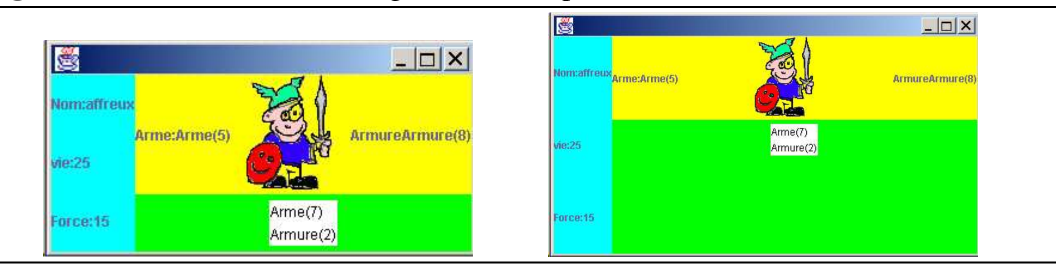


Figure 9.12 Instanciation des algorithmes de placement



Nous avons réalisé une autre gestion du placement. La figure 9.11 représente la nouvelle gestion du placement. Nous voulions passer “nom, force,vie” sur le coté droit et consacré la partie centrale de l’écran à l’affichage du reste des informations. Nous avons pour cela créé un nouveau noeud intermédiaire pour contenir la liste des équipements et l’affichage des armes et armures portées. Cet exemple montre bien comment en jouant sur l’arbre lui-même et les algorithmes de placement nous pouvons réaliser l’écran que nous voulons.

La figure 9.12 représente le résultat à l’écran. le code ci-dessous permet d’obtenir cet écran.

```
import java.util.*;
import javax.swing.*;
import java.awt.*;

class Main {
    public static void main(String args[]) {
        Mauhly m=new Mauhly("affreux",15,25);
        m.setArme(new Arme(5));
        m.setArmure(new Armure(8));
        m.addEquip(new Arme(7));
```

```
        m.addEquip(new Armure(2));

        // Fenêtre principale
        JFrame f=new JFrame();
        f.getContentPane().setBackground(Color.blue);
        f.getContentPane().setLayout(new BorderLayout());

        // Noeud pour Nom, vie, force ..
        JPanel p1=new JPanel();
        p1.setLayout(new GridLayout(0,1));
        p1.setBackground(Color.cyan);
        p1.add(new JLabel("Nom:"+m.getNom()));
        p1.add(new JLabel("vie:"+m.getVie()));
        p1.add(new JLabel("Force:"+m.getForce()));

        // Panel pour les armes et armure
        JPanel p2= new JPanel();
        p2.setLayout(new BorderLayout());
        p2.setBackground(Color.yellow);
        p2.add(new JLabel("Arme:"+m.getArme()),BorderLayout.WEST);
        p2.add(new JLabel(new ImageIcon("heros.gif")), BorderLayout.CENTER);
        p2.add(new JLabel("Armure"+m.getArmure()),BorderLayout.EAST);

        // panel pour visualiser le contenu du sac
        JPanel p3= new JPanel();
        p3.setBackground(Color.green);
        JList l=new JList(m.getEquip());
        p3.add(l);

        // aggrégation des différents panel
        f.getContentPane().add(p1,BorderLayout.WEST);
        JPanel center=new JPanel();
        center.setLayout(new BorderLayout());
        f.getContentPane().add(center,BorderLayout.CENTER);
        center.add(p2,BorderLayout.NORTH);
        center.add(p3,BorderLayout.CENTER);

        // Arbre prêt : Affichage
        f.pack();
        f.setVisible(true);
    }
}
```

9.3 Gérer les événements: le modèle par délégation

Le modèle par délégation est composé des éléments suivants:

- Des objets événements. Par exemple “KeyEvent”, “WindowEvent”, “ActionEvent”
- Des objets “producteur” d’événements. Par exemple, un bouton peut produire l’événement “ActionEvent”.

– Des objets “consommateur” d’événements. Ces objets s’abonnent aux événements en provenance des producteurs
– une queue d’événéments et un processeur chargé de propager les événéments vers les consommateurs.

Le comportement général du modèle par délégations est assez simple. les composants graphiques produisent des événements. les classes applicatives pour pouvoir être considérées comme consommateur d’événement doivent implanter les interfaces aqéquates. Par exemple, si une classe est intéréssée par des événements relatifs aux fenêtres, la classe applicative doit implanter “WindowsListener”. Si elle intéressée par des événements en provenance du clavier, elle doit implanter “KeyListener” . . . Implanter une interface implique de fournir l’implémentation des méthodes définie dans l’interface. Par exemple, si une classe applicative décide d’implanter l’interface “WindowListener”, elle devra fournir le code des méthodes suivantes:

```
package java.awt.event;
import java.util.EventListener;

public interface WindowListener extends EventListener {
    public void windowOpened(WindowEvent e);
    public void windowClosing(WindowEvent e);
    public void windowClosed(WindowEvent e);
    public void windowIconified(WindowEvent e);
    public void windowDeiconified(WindowEvent e);
    public void windowActivated(WindowEvent e);
    public void windowDeactivated(WindowEvent e);
}
```

Pour que les événements puissent être propagés des producteurs aux consommateurs, les consommateurs doivent s’abonner auprès des producteurs. Dans ce but, chaque producteur d’événements fournit des méthodes permettant aux consommateurs de s’inscrire dans leurs listes de diffusions. Par exemple, un objet fenêtre va fournir des méthodes “addWindowListener(WindowListener w)” et “removeWindowListener(WindowListener w)”. Un objet bouton va fournir “addActionListener(ActionListener w)” et “removeActionListener(ActionListener w) . . . .

On voit bien ici l’intérêt des interfaces. Le moteur de propagation d’événements peut être écrit sans pour autant connaître les classes applicatives qui vont recevoir ces événements. Il suffit de connaître les interfaces auxquelles ces classes applicatives devront se conformer. Ce type de construction est généralement appelé “framework”

Jusqu’à présent nous avons mis en place les producteurs d’événements en construisant les arbres de composants graphiques, nous pouvons créer les consommateurs d’événement en implantant les interfaces adéquates, nous pouvons enregistrer ces consommateurs auprès des producteurs, il ne manque plus qu’un acteur: le diffuseur d’événements.

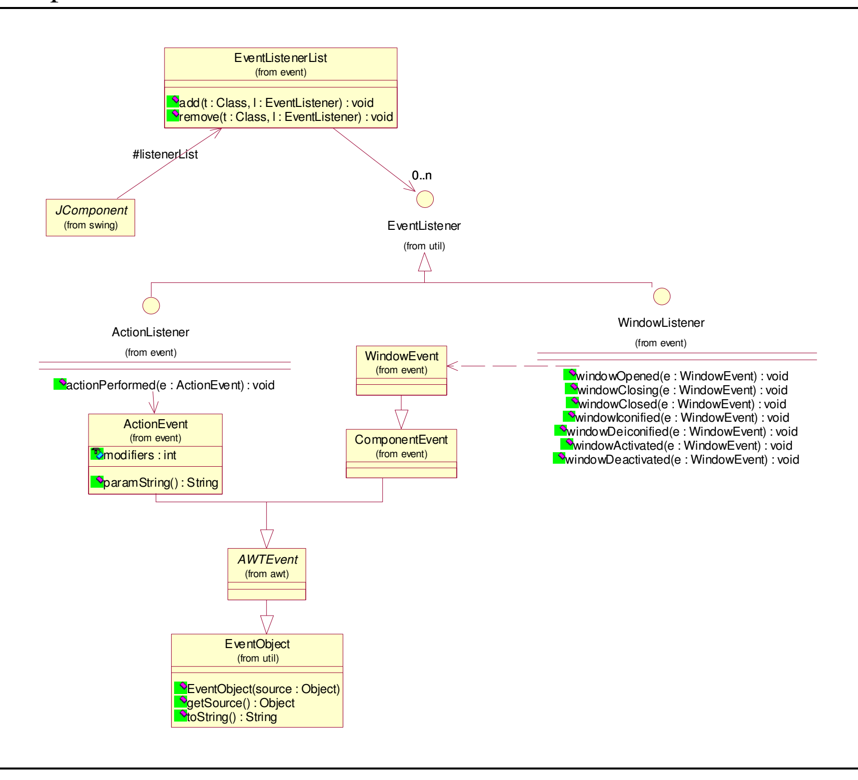
De manière très imagée, lorsqu’un événément est produit, il est stocké dans la queue d’événements. Un processus tournant dans la machine virtuelle (un thread) se charge de diffuser les événements. Cette queue d’événement permet de séparer la production d’événement de la diffusion d’événement. Très concrêtement, un autre événement peut être produit sans pour autant que le premier soit déjà diffusé. Il n’est donc pas nécessaire d’attendre qu’un événement soit complétement traité avant de pouvoir en traité un autre. Le diffuseur d’événement se base sur les déclaration de diffusion pour appeler sur les classes applicatives

les méthodes définies dans les interfaces i.e. “actionPermormed(ActionEvent e)”, “Mouse-Presses(MouseEvent e)”.

Le modèle par délégation est un modèle simple de gestion d’événement. Connaître son principe de fonctionnement permet de programmer en java sans pour autant connaître par coeur la librairie graphique. Il est en effet possible de déduire que forcément tel composant graphique doit fournir telles méthodes et qu’une interface de type “Listener” doit exister avec tel type de méthodes.

La figure 9.13 montre un fragment du digramme de classe de la librairie graphique.

Figure 9.13 producteurs et consommateurs d’événements



Le programme suivant montre une classe consommatrice de d’événements de type clavier, sourris, fenêtre .... Nous créons trois composants graphiques, une fenêtre, un bouton, une champ texte et nous déclarons les diffusion d’événements de la manière suivante:

– La fenêtre s’abonne aux événements de type souris, fenêtre et clavier.
– Le bouton et le champ texte s’abonne aux événements de type “action” (bouton appuyé ou modification de texte validée).

La figure 9.14 montre le rendu graphique du programme. Nous montrons Ci-dessous le texte imprimée sur la sortie standard:

```
Received event: java.awt.event.WindowEvent[WINDOW_ACTIVATED] on frame0
Received event: java.awt.event.FocusEvent[FOCUS_GAINED,permanent] on javax.swing.JButton[,10,5,75x27,...]
Received event: java.awt.event.WindowEvent[WINDOW_OPENED] on frame0
Received event: java.awt.event.MouseEvent[MOUSE_MOVED,(64,64),mods=0,clickCount=0] on frame0
Received event: java.awt.event.KeyEvent[KEY_PRESSED,keyCode=18,keyChar='?',modifi
ers=Alt] on frame0
```

```
Received event: java.awt.event.KeyEvent[KEY_RELEASED,keyCode=154,Print Screen,mod
 ifiers =Alt] on frame0
Received event: java.awt.event.KeyEvent[KEY_RELEASED,keyCode=18,keyChar='?'] on f
rame0
```

Figure 9.14 Un exemple simple




```
import java.util.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class EventCatcher implements
    ActionListener,WindowListener, KeyListener, FocusListener, MouseMotionListener {

    private void msg(AWTEvent e) {
        System.out.println("Received event:"+e);
    }

    // from ActionListener
    public void actionPerformed(ActionEvent e) { msg(e); }

    // from KeyListener
    public void keyTyped(KeyEvent e) { msg(e); }
    public void keyPressed(KeyEvent e) { msg(e); }
    public void keyReleased(KeyEvent e) { msg(e); }

    // from WindowListener
    public void windowOpened(WindowEvent e) { msg(e); }
    public void windowClosing(WindowEvent e) { msg(e);}
    public void windowClosed(WindowEvent e) { msg(e);}
    public void windowIconified(WindowEvent e) { msg(e);}
    public void windowDeiconified(WindowEvent e){ msg(e);}
    public void windowActivated(WindowEvent e){ msg(e);}
    public void windowDeactivated(WindowEvent e){msg(e);}

    public void focusGained(FocusEvent e){msg(e);}
    public void focusLost(FocusEvent e){msg(e);}

    public void mouseDragged(MouseEvent e){msg(e);}
    public void mouseMoved(MouseEvent e){msg(e);}
}
```

```
class Main {
    public static void main(String args []) {
        JFrame jf=new JFrame();
        jf.getContentPane().setLayout(new FlowLayout());

        EventCatcher ev=new EventCatcher();

        jf.addWindowListener(ev);
        jf.addKeyListener(ev);
        jf.addMouseMotionListener(ev);
        JButton jb=new JButton("appuie␣ici");
        jb.addActionListener(ev);
        jb.addFocusListener(ev);
        JTextField jtf=new JTextField("écrit␣ici␣!");
        jtf.addActionListener(ev);

        jf.getContentPane().add(jb);
        jf.getContentPane().add(jtf);

        jf.pack();
        jf.setVisible(true);
    }
}
```

Vous remarquez dans l'exemple ci-dessus que tous les événements sont propagés vers une unique instance d'"eventCatcher". Nous pouvons tout à fait créer plusieurs consommateur d'événements. Nous pouvons modifier le programme de la manière suivante:

```
import java.util.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class EventCatcher implements
    ActionListener,WindowListener, KeyListener, FocusListener, MouseMotionListener {

    private String name;

    EventCatcher(String name) {
        this.name=name;
    }

    private void msg(AWTEvent e) {
        System.out.println(name+"␣Received␣event:"+e);
    }

    // from ActionListener
    public void actionPerformed(ActionEvent e) { msg(e); }

    // from KeyListener
    public void keyTyped(KeyEvent e) { msg(e); }
    public void keyPressed(KeyEvent e) { msg(e); }
    public void keyReleased(KeyEvent e) { msg(e); }
```

```
    // from WindowListener
    public void windowOpened(WindowEvent e) { msg(e); }
    public void windowClosing(WindowEvent e) { msg(e);}
    public void windowClosed(WindowEvent e) { msg(e);}
    public void windowIconified(WindowEvent e) { msg(e);}
    public void windowDeiconified(WindowEvent e){ msg(e);}
    public void windowActivated(WindowEvent e){ msg(e);}
    public void windowDeactivated(WindowEvent e){msg(e);}

    public void focusGained(FocusEvent e){msg(e);}
    public void focusLost(FocusEvent e){msg(e);}

    public void mouseDragged(MouseEvent e){msg(e);}
    public void mouseMoved(MouseEvent e){msg(e);}
}

class Main {
    public static void main(String args []) {
        JFrame jf=new JFrame();
        jf.getContentPane().setLayout(new FlowLayout());

        EventCatcher ev1=new EventCatcher("ev1");
        EventCatcher ev2=new EventCatcher("ev2");


        jf.addWindowListener(ev1);
        jf.addKeyListener(ev1);
        jf.addMouseMotionListener(ev1);
        JButton jb=new JButton("Bouton");
        jb.addActionListener(ev2);
        jb.addFocusListener(ev2);
        JTextField jtf=new JTextField("Texte");
        jtf.addActionListener(ev2);

        jf.getContentPane().add(jb);
        jf.getContentPane().add(jtf);

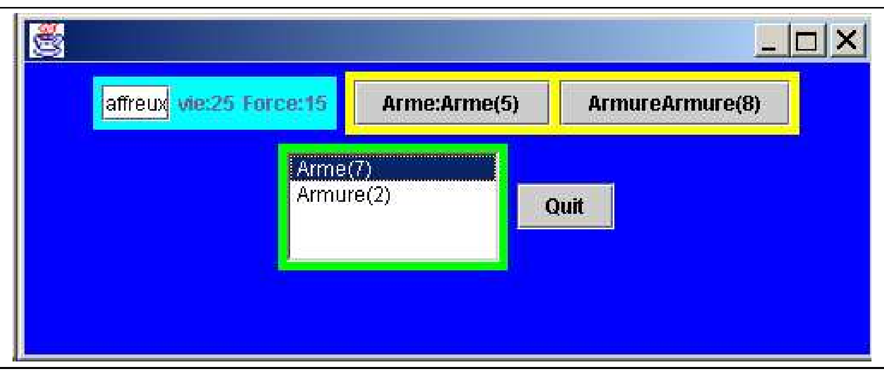
        jf.pack();
        jf.setVisible(true);
    }
}
```

```
ev1 Received event: java.awt.event.WindowEvent[WINDOW_ACTIVATED] on frame0
ev2 Received event: java.awt.event.FocusEvent[FOCUS_GAINED,permanent] on javax.swing.JButton...
ev1 Received event: java.awt.event.WindowEvent[WINDOW_OPENED] on frame0
ev1 Received event: java.awt.event.MouseEvent[MOUSE_MOVED,(28,30),mods=0,clickCou
nt=0] on frame0
ev1 Received event: java.awt.event.WindowEvent[WINDOW_DEACTIVATED] on frame0
ev2 Received event: java.awt.event.FocusEvent[FOCUS_LOST,temporary] on javax.swin
```

9.3.1 Un exemple simple

Nous appliquons le modèle par délégation sur notre exemple de visualisation de l'état du héros. L'écran de visualisation a quelque peut changé. La figure 9.15 représente notre nouvel écran. Un bouton “Quit” a été ajouté pour permettre de quitter l'application. Le label pour visualiser le nom du joueur a été remplacé par un “textfield” permettant de changer le nom du joueur.

Figure 9.15 Ecran de visualisation



La figure 9.16 montre comment nous gérons le changement de nom du joueur. La figure 9.18 montre comment nous gérons le bouton “Quit”.

Figure 9.16 Diagramme de classe pour la gestion du nom

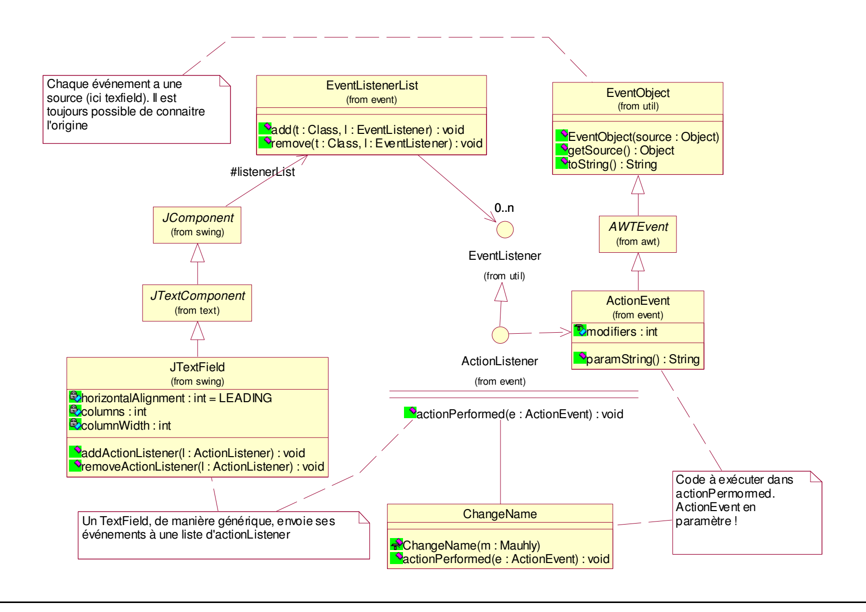


Figure 9.17 Diagramme de classe pour la gestion du bouton “quit”

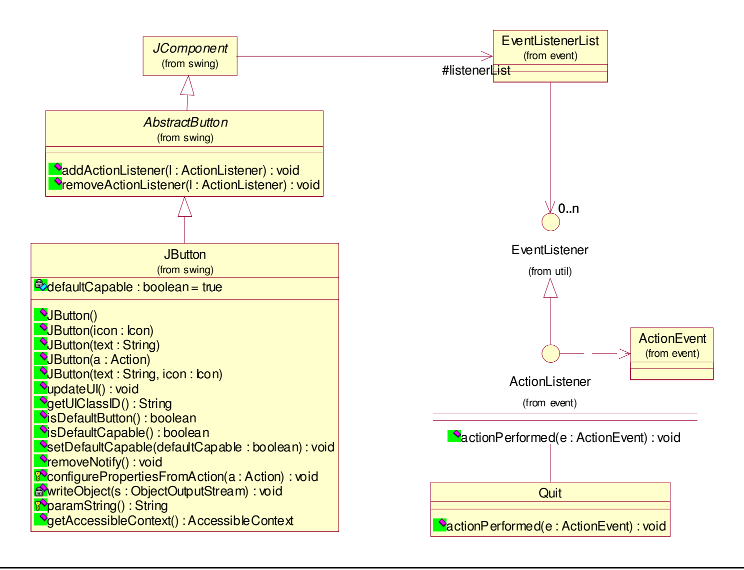


Figure 9.18 Diagramme d’objets pour l’application

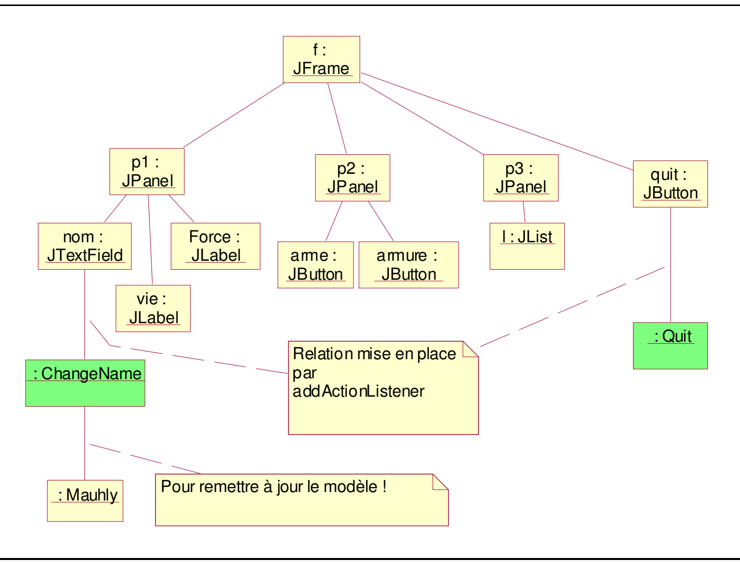


Figure 9.19 Diagramme de séquence de mise en place

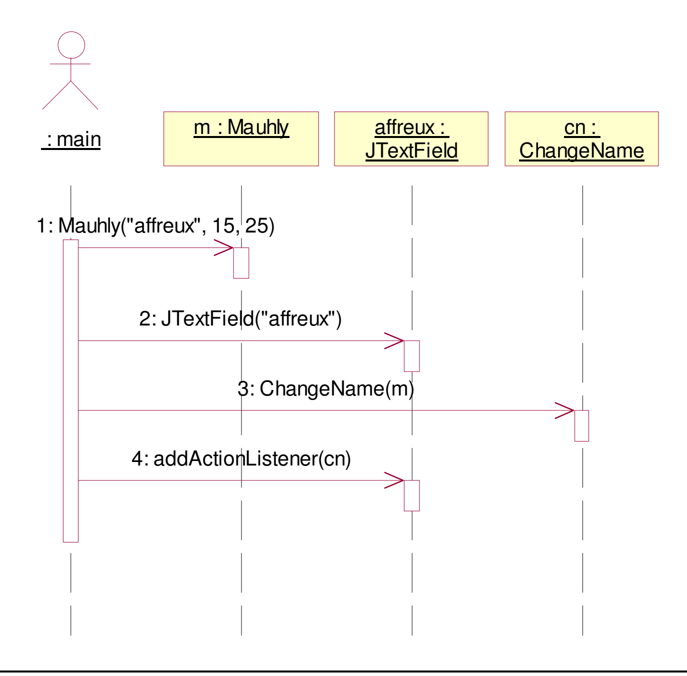
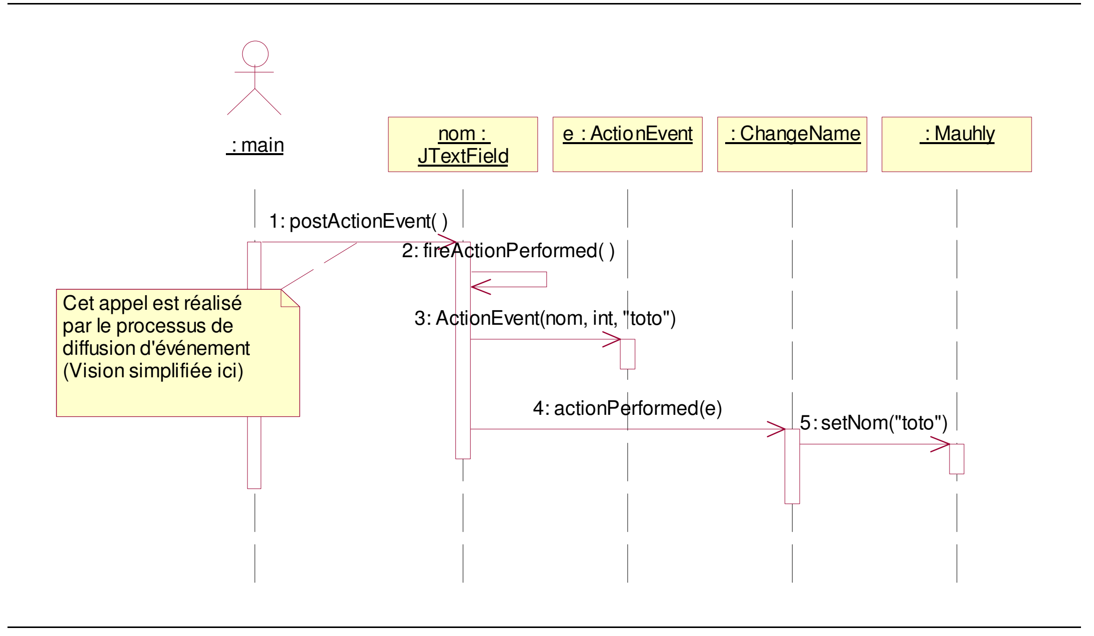


Figure 9.20 Diagramme simplifié d'activation



Voici le code complet du programme.

```
import java. util .*;
import javax.swing.*;
import java.awt.*;
import java.awt.event .*;
```

```
class Quit implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.exit (0);
    }
}

class ChangeName implements ActionListener {
    private Mauhly m;
    ChangeName(Mauhly m) {
        this.m=m;
    }
    public void actionPerformed(ActionEvent e) {
        System.out.println (e.paramString ());
        System.out.println (e.getActionCommand());
        m.setNom(e.paramString ());
    }
}

class Main {
    public static void main(String args []) {
        Mauhly m=new Mauhly("affreux",15,25);
        m.setArme(new Arme(5));
        m.setArmure(new Armure(8));
        m.addEquip(new Arme(7));
        m.addEquip(new Armure(2));

        // Fenêtre principale
        JFrame f=new JFrame();
        f.getContentPane().setBackground(Color.blue);
        f.getContentPane().setLayout(new FlowLayout());

        // Noeud pour Nom, vie, force ..
        JPanel p1=new JPanel();
        p1.setBackground(Color.cyan);
        JTextField jnom;
        p1.add(jnom=new JTextField(m.getNom()));
        jnom.addActionListener(new ChangeName(m));

        p1.add(new JLabel("vie:"+m.getVie ()));
        p1.add(new JLabel("Force:"+m.getForce ()));

        // Panel pour les armes et armure
        JPanel p2= new JPanel();
        p2.setBackground(Color.yellow);
        JButton jarme, jarmure;
        p2.add(jarme=new JButton("Arme:"+m.getArme()));
        p2.add(jarmure=new JButton("Armure"+m.getArmure()));

        // panel pour visualiser le contenu du sac
        JPanel p3= new JPanel();
        p3.setBackground(Color.green);
        java.awt.List l=new java.awt.List ();
```

```
        for (Enumeration e = m.getEquip() ; e.hasMoreElements() ;) {
            l.add("␣"+e.nextElement());
        }
        p3.add(l);


        // aggrégation des différents panel
        f.getContentPane().add(p1);
        f.getContentPane().add(p2);
        f.getContentPane().add(p3);

        // Quitter
        JButton jquit =new JButton("Quit");
        jquit.addActionListener(new Quit());
        f.getContentPane().add(jquit);

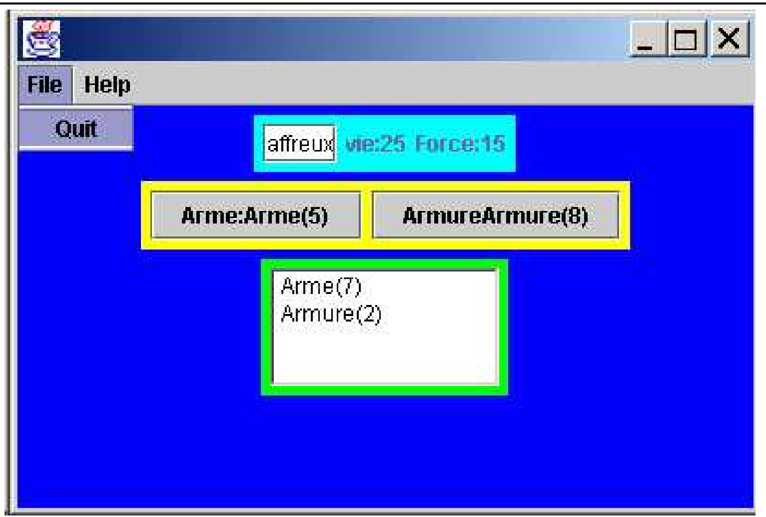

        // Arbre prêt : Affichage
        f.pack();
        f.setVisible(true);
    }
}
```

9.3.2 Gestion des menus

La gestion des menus reste tout à fait dans l'esprit du modèle par composition pour la construction des menus et du modèle par délégation pour la gestion des événements. La figure 9.21 l'écran de visualisation de l'état du héros avec des menus.

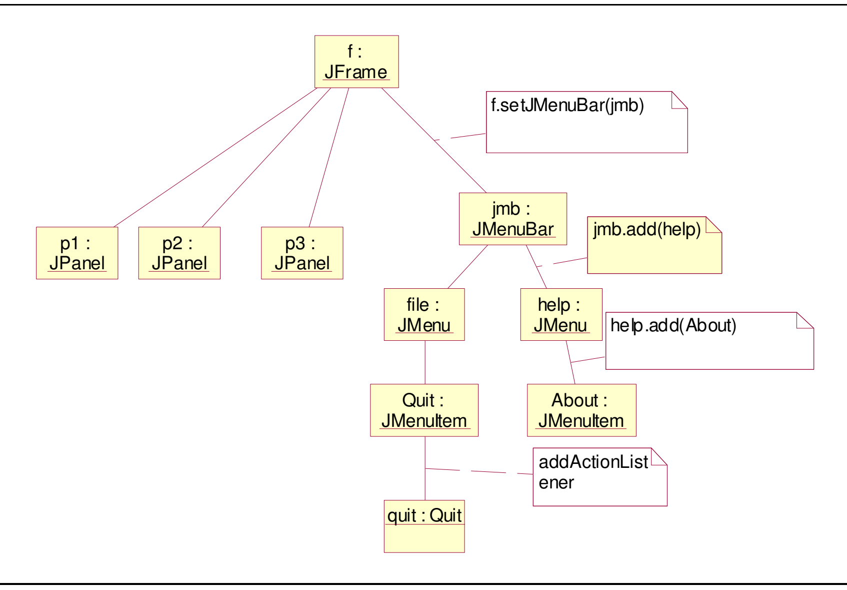


Figure 9.21 Diagramme simplifié d'activation



Pour construire les menus, nous avons construits un arbre de composant avec des nouveaux composants graphiques: JMenuBar, JMenu, JMenuItem. La figure **??** montre l'arbre graphique de cette application.

Figure 9.22 Diagramme d'objet d'un écran graphique avec menus



Le code ci-dessous montre le code ayant permis de créer cet écran. Dans ce code, la séléction de “file/quit” dans le menu permet de terminer l'application.

```
import java.util.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;


class Quit implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
}

class Main {
    public static void main(String args[]) {
        Mauhly m=new Mauhly("affreux",15,25);
        m.setArme(new Arme(5));
        m.setArmure(new Armure(8));
        m.addEquip(new Arme(7));
        m.addEquip(new Armure(2));

        // Fenêtre principale
        JFrame f=new JFrame();
        f.getContentPane().setBackground(Color.blue);
        f.getContentPane().setLayout(new FlowLayout());

        JMenuBar jmb=new JMenuBar();
        JMenu jmfile=new JMenu("File");
        JMenuItem quit=null;
        jmfile.add(quit=new JMenuItem("Quit"));
        quit.addActionListener(new Quit());
```

```
        JMenu jmhelp=new JMenu("Help");
        jmhelp.add(new JMenuItem("About"));
        jmb.add( jmfile );
        jmb.add(jmhelp);
        f.setJMenuBar(jmb);

        // Noeud pour Nom, vie, force ..
        JPanel p1=new JPanel();
        p1.setBackground(Color.cyan);
        JTextField jnom;
        p1.add(jnom=new JTextField(m.getNom()));
        jnom.addActionListener(new ChangeName(m));

        p1.add(new JLabel("vie:"+m.getVie()));
        p1.add(new JLabel("Force:"+m.getForce()));

        // Panel pour les armes et armure
        JPanel p2= new JPanel();
        p2.setBackground(Color.yellow);
        JButton jarme, jarmure;
        p2.add(jarme=new JButton("Arme:"+m.getArme()));
        p2.add(jarmure=new JButton("Armure"+m.getArmure()));

        // panel pour visualiser le contenu du sac
        JPanel p3= new JPanel();
        p3.setBackground(Color.green);
        java.awt.List l=new java.awt.List();
        for (Enumeration e = m.getEquip() ; e.hasMoreElements() ;) {
            l.add(" "+e.nextElement());
        }
        p3.add(l);


        // aggrégation des différents panel
        f.getContentPane().add(p1);
        f.getContentPane().add(p2);
        f.getContentPane().add(p3);

        // Arbre prêt : Affichage
        f.pack();
        f.setVisible(true);
    }
}
```

9.3.3 Gestion des dialogues

Construire un dialogue consiste à créer des nouvelles fenêtres dont l'objectifs est d'informer l'utilisateur, de poser une question à l'utilisateur ou de demander à l'utilisateur de saisir une variable ou un texte. Deux caractéristiques essentielles différencient un fenêtre de dialogue d'une fenêtre normale:

– Une fenêtre de dialogue ne peux exister sans une fenêtre mère.

– Une fenêtre de dialogue est “modale”. La modalité d'un dialogue définie si l'utilisateur a la possibilité d'intéragir avec les autres fenêtres quand la fenêtre de dialogue est affichée. La modalité est donc une valeur booléene. Si le dialogue est modal, alors l'utilisateur ne peut intéragir qu'avec la fenêtre de dialogue. Sinon, l'utilisateur peut intéragir avec toutes fenêtres de l'application.

Nous avons ajouté deux dialogue à notre écran de visualisation de l'état du héro. Avant de quitter, l'application demande une configuration et le menu “help/about” donne des informations à propos de l'application. La figure 9.23 représente ces dialogues.

Figure 9.23 Dialogues




```
import java.util.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;


class Quit implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
}

class CloseWindow implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        Window w=(Window)(((JComponent)e.getSource()).getTopLevelAncestor());
        w.setVisible(false);
    }
}

class QuitDialog implements ActionListener {
    private JDialog jd;

    QuitDialog(JFrame f) {
        jd=new JDialog(f,"Quit",true);
        jd.getContentPane().setLayout(new BorderLayout());
        jd.getContentPane().add(BorderLayout.CENTER,new JLabel("Voulez-vous vraiment sortir"));
        JButton oui=new JButton("oui");
        JButton non=new JButton("non");
        oui.addActionListener(new Quit());
        non.addActionListener(new CloseWindow());
        JPanel jp=new JPanel();
        jp.add(oui);
```

```
        jp.add(non);
        jd.getContentPane().add(BorderLayout.SOUTH,jp);
        jd.pack();
    }

    public void actionPerformed(ActionEvent e) {
        jd.setVisible(true);
    }
}

class About implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        Frame f=(Frame)(((JComponent)e.getSource()).getTopLevelAncestor());
        JOptionPane.showMessageDialog(f,"Tortues JAVA , MARS 2001",
                                      "Tortues Java",
                                      JOptionPane.INFORMATION_MESSAGE);
    }
}

class Main {
    public static void main(String args[]) {
        Mauhly m=new Mauhly("affreux",15,25);
        m.setArme(new Arme(5));
        m.setArmure(new Armure(8));
        m.addEquip(new Arme(7));
        m.addEquip(new Armure(2));

        // Fenêtre principale
        JFrame f=new JFrame();
        f.getContentPane().setBackground(Color.blue);
        f.getContentPane().setLayout(new FlowLayout());

        JMenuBar jmb=new JMenuBar();
        JMenu jmfile=new JMenu("File");
        JMenuItem quit=null;
        jmfile.add(quit=new JMenuItem("Quit"));
        quit.addActionListener(new QuitDialog(f));
        JMenu jmhelp=new JMenu("Help");
        JMenuItem about=null;
        jmhelp.add(about=new JMenuItem("About"));
        about.addActionListener(new About());
        jmb.add(jmfile);
        jmb.add(jmhelp);
        f.setJMenuBar(jmb);

        // Noeud pour Nom, vie, force ..
        JPanel p1=new JPanel();
        p1.setBackground(Color.cyan);
        JTextField jnom;
        p1.add(jnom=new JTextField(m.getNom()));
        jnom.addActionListener(new ChangeName(m));

        p1.add(new JLabel("vie:"+m.getVie()));
```

```
        p1.add(new JLabel("Force:"+m.getForce ()));

        // Panel pour les armes et armure
        JPanel p2= new JPanel ();
        p2.setBackground(Color.yellow);
        JButton jarme, jarmure;
        p2.add(jarme=new JButton("Arme:"+m.getArme()));
        p2.add(jarmure=new JButton("Armure"+m.getArmure()));

        // panel pour visualiser le contenu du sac
        JPanel p3= new JPanel ();
        p3.setBackground(Color.green);
        java.awt.List l=new java.awt.List ();
        for (Enumeration e = m.getEquip() ; e.hasMoreElements() ;) {
            l.add("␣"+e.nextElement ());
        }
        p3.add(l);


        // aggrégation des différents panel
        f.getContentPane().add(p1);
        f.getContentPane().add(p2);
        f.getContentPane().add(p3);

        // Arbre prêt : Affichage
        f.pack();
        f.setVisible(true);
    }
}
```

9.4 Applications graphiques et classes internes

Le modèle par délégation s'implante mal en Java. En fait, le modèle “on Event do Action” est difficilement conciliable avec un approche objet. En effet, “do Action” est typiquement une opération au sens classique et non une méthode d'un classe. Si on applique le modèle par délégation tel quel en java, on se retrouve confronté la mise en place de classes artificielles juste pour héberger une méthode “Action”.

Figure 9.24 Modèle par délégation et classes artificielles

```
import java.awt.*;
import java.awt.event.*;

class Quit implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
}

class Main {
    public static void main(String args[]) {
        Frame f=new Frame();
        Button b;
        f.add(b=new Button("quit"));
        b.addActionListener(new Quit());
        f.pack();
        f.setVisible(true);
    }
}
```

La figure 9.4 illustre bien ce problème, nous voyons bien ici que la classe Quit ne sert qu'à hébérger la méthode $actionPerformed$. Elle ne sert qu'à fournir un receveur factice pour l'appel de la méthode actionPerformed. La création d'un objet "Quit" ligne 15 n'a que peu de sens.

Les classes internes peuvent fournir une solution plus élégante à ce problème comme nous le montrerons un peu plus tard. Pour l'instant nous nous intéressons aux classes internes comme nouvelle fonctionnalité du langage.

9.4.1 Classes et objets internes, Classes et objets externes

Il est possible en Java de définir des classes à l'intérieur d'une classe. Nous avons choisit de présenter cette fonctionnalité du langage maintenant que nous disposons d'un problème pertinent pour utilisation. (cf Inner class Specification).

En Java il est possible de:

- déclarer une classe dans une classe
- déclarer une classe dans une méthode
- déclarer une classe anonyme dans une méthode

Pour fixer le vocabulaire, nous prenons la convention suivante: si une classe A est déclarée à l'intérieur B d'une classe B, alors A est une classe interne et B est sa classe externe.

La conséquence pratique de telle définitions est la suivante:

- Un objet instance d'une classe interne ne peut exister sans une instance de sa classe externe. Par abus de langage, nous parlerons d'objets internes et externes.
- Un object interne peut observer directement l'état de son objet externe, même si cet état est privé.

Prenons l'exemple de la carte des tortues java. Nous voulons contruire une représentation graphique de cette carte. Nous voulons donc contruire un objet graphique qui représente cette

carte. Pour mettre en place cet objet graphique il faut accéder à l'état de l'objet carte. Bien sur, nous pouvons créer une classe CarteGraphique avec une relation 1-1 avec la classe carte, nous pouvons aussi considérer cet objet graphique comme le représentant graphique de la carte et à ce titre le déclarer comme instance d'une classe interne de Carte.

La différence entre les deux solutions n'est pas très grande d'un point de vue opérationel.

Figure 9.25 Utilisation des classes internes

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class Carte {
    private Region regions [][];

    public Carte(int nbrow, int nbcol) {
        regions=new Region[nbrow][nbcol];
        for (int i=0;i<regions.length;i++) {
            for (int j=0;j<regions[i].length;j++) {
                regions[i][j]=new Region(this,i,j);
            }
        }
    }

    public Region getRegion(int x, int y) {
        try {
            return regions[x][y];
        } catch (ArrayIndexOutOfBoundsException e) {
            return null;
        }
    }

    public JComponent display() {
        JComponent c=new IHM();
        return c;
    }

    class IHM extends JPanel {

        public IHM() {
            this.setLayout(new GridLayout(regions.length, regions[1].length));
            for (int i=0;i<regions.length;i++) {
                for (int j=0;j<regions[i].length;j++) {
                    this.add(new JLabel("Case:"+i+","+j));
                }
            }
        }
    }
}
```

9.5 Principe du MVC

Le modèle par délégation permet de propager les événements vers les classes applicatives. Les objets receptionnant les événements provoquent les changements d'état du modèle. Ces objets recepteurs en controlent en quelque sorte les changements d'état sur le modèle. On appelle ces objets les controleurs.

Le problème est maintenant le suivant, quand le modèle change d'état, comment propager le nouvel état vers les composants graphiques qui visualisent le modèle.

Nous avons construit une nouvelle version du visualisateur d'état du héros basée sur le MVC. La figure 9.26 visualise ce nouvel écran. La figure 9.27 montre ce même écran après que le héros se soit reposé plusieurs fois et enlevé son armure.

Figure 9.26 Un écran de visualisation basée sur le MVC

Arme:Arme(5) ArmureArmure(8)
Nom:affreux
Vie:25
Force:15
Arme(7)
Armure(2)
Repos
Enlever Arme
Enlever Armure
affreux
Vie:25
Force:15

Figure 9.27 Propagation des changements d'état du modèle vers les vues

Cet écran est le résultat de l'instanciation du diagramme de classe de la figure 9.28. La figure 9.29 représente le diagramme d'objets de cet écran.

Dans le MVC, les rôles à la base sont bien définis:

– La vue est un objet graphique ou une composition d'objets graphique. Sa caractéristique principale et de pouvoir recevoir des notifications du modèle que celui-ci change d'état. Pour que le modèle puissent envoyer des messages à une vue, celle-ci doit se conformer à une interface. Dans la librairie Java, cette interface est java.util.Observer. Dans la figure 9.28, les vues sont les classes implémant l'interface java.util.Observer. Elles doivent donc implémenter la méthode "update(...)".

  Pour remettre les composants graphiques à jour, les vues doivent pouvoir accéder au modèle et relire l'état du modèle. Pour cela, elles doivent avoir une référence sur le modèle. Cette référence peut-être obtenue en utilisant le premier paramètre de la méthode "update" qui référence l'objet "observable" qui a émis la notification. Cette référence peut gérée au moment de la construction, que on construit une vue, on donne en paramètre du construction quel est le modèle observé. Ici, nous avons choisit de déclarer les vues comme classes internes du modèle. Cela a plusieurs conséquences:

  1. La vue ne peut exister sans le modèle. En effet, une instance d'une classe interne ne peut exister sans une instance dans sa classe externe.
  2. La vue a un accès privilégié à l'état du modèle. En effet, un objet interne voit directement l'état de son objet externe. Cet accès priviliégié peut faciliter la remise à jour de la vue lors des notifications.
  3. Par construction, c'est le modèle qui doit fournir une méthode pour construire la vue. "Afficher" une vue fait alors partie du comportement du modèle. Cette façon de faire peut-être restrictive. Il est parfois plus facile de construire séparément les vues et le modèle et gérer les attachements par la suite.

- Le modèle représente l'objet ou les objets applicatifs. Dans notre exemple, la classe "Maulhy" joue le rôle du modèle. Un modèle peut avoir 0 ou n vues qui lui sont associées. Cela signifie deux choses:
  1. Quand une vue est créée, elle doit s'enregistrer auprès de son modèle. Le modèle doit donc fournir une méthode du type "addObserver(Observer o)".
  2. Quand le modèle change d'état, il doit appeller sur l'ensemble des vues enregistrées, la méthode "update()".

  Un modèle est donc un objet abstrait gérant une liste de vues. Il peut donc être représenté par une classe abstraite. La classe java.util.Observable joue ce rôle. Dans la figure 9.28, la classe "Mauhly", notre modèle hérite de "Observable" et donc hérite des méthode "addObserver", "notifyObserver" ....
- Le controleur provoque les changements d'états sur le modèle. Conrêtement, c'est lui qui appelle les méthodes de changements d'état. Dans notre exemple, c'est en se reposant et ou enlevant armes et armures que le modèle change. C'est donc les boutons "repos", "enlever arme" et "enlever armure" qui vont provoquer les changements dans le modèle. Pour être plus précis, ce sont les classes qui écoutent les événements en provenance de ces boutons qui vont réellement opérer le changement d'état du modèle.

  Dans notre exemple, le contrôleur est représenté par la classe "Controller". Cette classe est elle-même une vue. Ceci n'est pas obligatoire mais arrive très fréquemment. En effet, l'état des boutons peut changer suivant l'état du modèle. Par exemple, une fois qu'on a enlevé l'armure, le bouton permettant d'enlever l'armure se grise automatiquement. Cela est du au fait que le changement d'état corresposant à la remise dans l'armure dans le sac va être notifié au contrôleur qui va grise le bouton correspondant.

  Le contrôleur doit avoir une référence sur le modèle pour pouvoir appeller les méthodes de changement d'état. Dans notre exemple, nous avons déclaré le contrôleur comme classe interne du modèle. Ceci ne fait pas partie du MVC originel, mais quelques petits avantages: (1) par construction, le contrôleur ne peut pas exister sans le modèle, (2) le contrôleur peut observer plus facilement l'état du modèle (3) le code du contôleur se trouve physiquement dans le même fichier que le code du modèle mais en restant bien conditionné à part dans une classe interne.

  Cette construction en utilisant les classes internes n'est pas obligatoire.

Figure 9.28 Modèle, Vue, Controleur

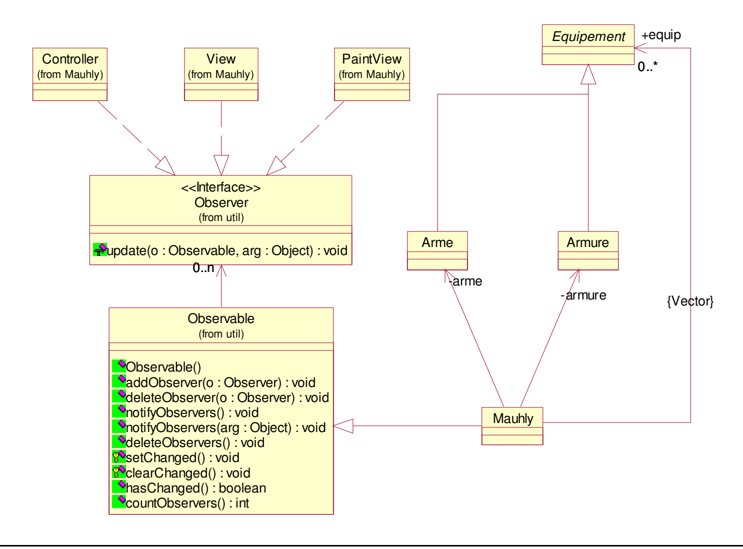
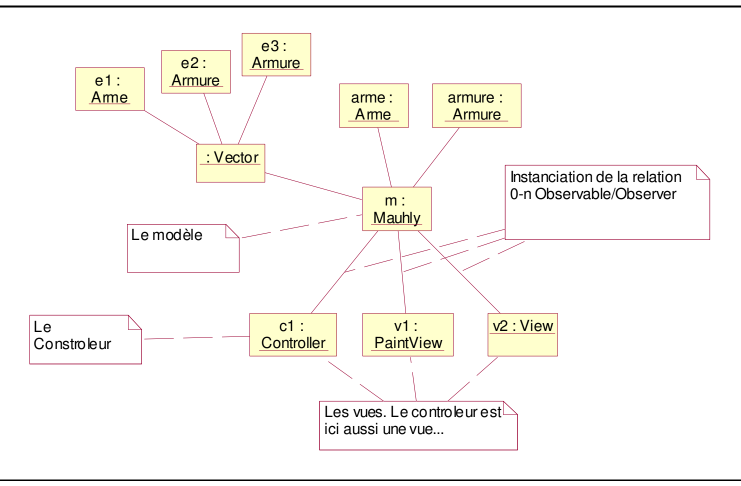


Figure 9.29 Modèle, Vue, Controleur




```
import javax.swing.*;
import java.awt.event.*;
import java.util.*;
import java.awt.*;

class Mauhly extends Observable {
    private String nom;
    private int force;
    private int vie;
    private int exp=10;

    private Arme arme;
    private Armure armure;
    private Vector equip=new Vector();
```

```
    public Mauhly(String nomp, int forcep, int viep) {
        nom=nomp;
        force=forcep;
        vie=viep;
    }

    public void setArme(Arme arme) {
        this.arme=arme;
        this.setChanged();
        notifyObservers();
    }

    public void unsetArme() {
        this.addEquip(this.getArme());
        this.setArme(null);
    }

  // ...

    public Component View() {
        return new View();
    }
    public Component paintView() {
        return new PaintView();
    }

    class Controller extends JPanel implements Observer {
        private JButton jrepos= new JButton("Repos");
        private JButton jarme= new JButton("Enlever␣Arme");
        private JButton jarmure= new JButton("Enlever␣Armure");

        Controller() {
            this.setLayout(new GridLayout(0,2));
            Mauhly.this.addObserver(this);
            jrepos.addActionListener(new ActionListener() {
                    public void actionPerformed(ActionEvent e) {
                        Mauhly.this.reposer();
                    }
                });

            jarme.addActionListener(new ActionListener() {
                    public void actionPerformed(ActionEvent e) {
                        Mauhly.this.unsetArme();
                    }
                });

            jarmure.addActionListener(new ActionListener() {
                    public void actionPerformed(ActionEvent e) {
                        Mauhly.this.unsetArmure();
                    }
                });
            this.add(jrepos);
```

```
        this .add(jarme);
        this .add(jarmure);
    }

    public void update(Observable o, Object arg) {
        jarme. setEnabled(Mauhly.this.arme!=null);
        jarmure. setEnabled(Mauhly.this.armure!=null);
    }
}

class View extends JPanel implements Observer {
    JPanel p1=new JPanel();
    JPanel p2=new JPanel();
    JPanel p3=new JPanel();
    JLabel jname=new JLabel("Nom:"+Mauhly.this.nom);
    JLabel jvie=new JLabel("Vie:"+Mauhly.this.vie);
    JLabel jforce=new JLabel("Force:"+Mauhly.this.force);
    JLabel jarme=new JLabel("Arme:"+Mauhly.this.arme);
    JLabel jarmure=new JLabel("Armure"+Mauhly.this.armure);
    JList jsac=new JList(Mauhly.this.equip);

    View() {
        Mauhly.this.addObserver(this);

        this.setLayout(new BorderLayout());
        p1.setBackground(Color.cyan);
        p1.setLayout(new GridLayout(0,1));
        p1.add(jname);
        p1.add(jvie);
        p1.add(jforce);

        // Panel pour les armes et armure
        p2.setBackground(Color.yellow);
        p2.add(jarme);
        p2.add(jarmure);

        // panel pour visualiser le contenu du sac
        p3.add(jsac);

        this.add(p1,BorderLayout.WEST);
        this.add(p2,BorderLayout.NORTH);
        this.add(p3,BorderLayout.CENTER);
        this.add(new Mauhly.Controller(), BorderLayout.SOUTH);
    }

    public void update(Observable o, Object arg) {
        jname.setText("Nom:"+Mauhly.this.nom);
        jvie.setText("Vie:"+Mauhly.this.vie);
        jforce.setText("Force:"+Mauhly.this.force);
        jarme.setText("Arme:"+Mauhly.this.arme);
        jarmure.setText("Armure"+Mauhly.this.armure);
        jsac.setListData(Mauhly.this.equip);
    }
```

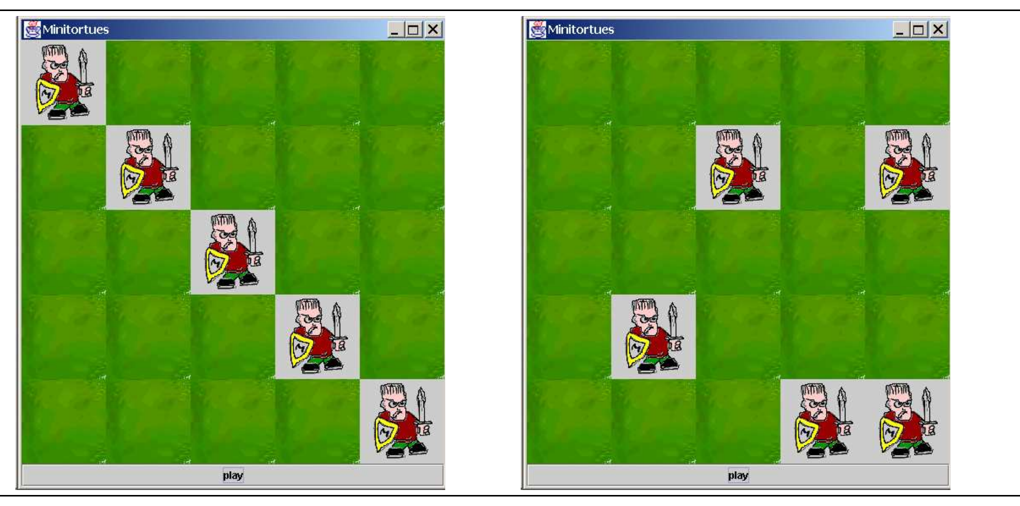
```
    }

    class PaintView extends Canvas implements Observer {
        // ...
    }
}
```

9.5.1 Minitortues: le MVC en action

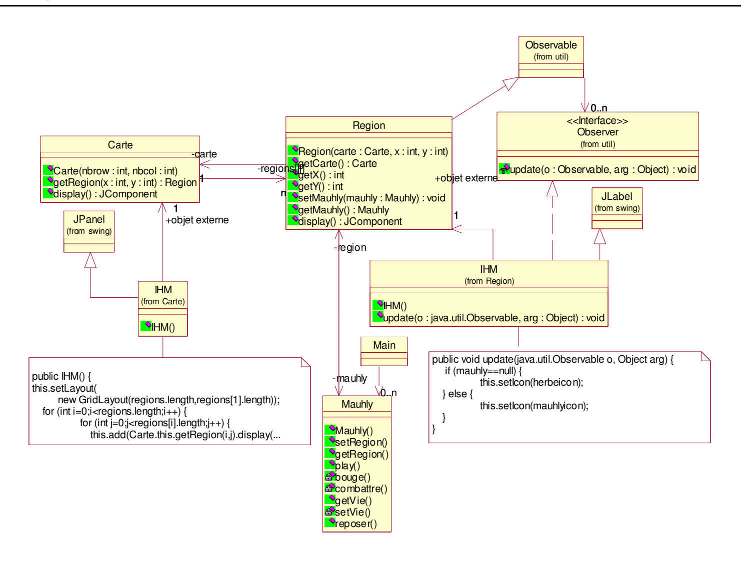
Nous avons appliqué le principe du MVC pour construire un embryon du jeu des tortues java. Nous avons construit un programme où des monstres peuvent combattre entre eux sur une carte. La figure 9.30 montre le résultat graphique. Les monstres dans ce cas applique à chaque tour une stratégie préféfinie.

Figure 9.30 Minitortues



La figure 9.31 représente le diagramme de classe de notre application. Si vous avez bien compris le MVC vous pouvez facilement reconnaitre les différent élément sur ce diagramme. Nous avons considéré que la région constitue le modèle. En effet, les monstres se déplacent de région en région en modifiant à chaque l'état de la région orgine et de la région cible. Si l'on applique le MVC à la région, chaque fois qu'un région change d'état, elle va avertir sa ou ses vues qui vont revenir lire l'état du modèle et se modifier en conséquence. La vue du modèle est représentée par une classe interne de la classe région.

Figure 9.31 Diagramme de classe des minitortues



Le code suivant constitue le programme principal de notre application “minitortue”. Le canvas est trivial: on crée la carte, on crée les monstres sur la carte, on demande à la carte de se dessiner, la carte en recevant ce message demande à chacune de ses régions de dessiner. L’écran de jeu est en place.

Chaque fois que l’utilisateur appuie sur le bouton “play”, les monstres se déplacent et éventuellement combattent. En se déplaçant, les monstrent modifient l’état des régions concernées qui signalent ce changement d’état à leurs vues respectivent qui se remettent ainsi à jour.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class Main {
    public static void main(String args[]) {
        Carte carte=new Carte(5,5);
        final Mauhly monstres[]=new Mauhly[5];
        for (int i=0;i<monstres.length;i++) {
            monstres[i]=new Mauhly("momo"+i,10,10,carte.getRegion(i,i));
        }

        JFrame frame=new JFrame("Minitortues");

        frame.getContentPane().add(carte.display(), BorderLayout.CENTER);

        JButton jb=new JButton("play");
        jb.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    for (int j=0;j<monstres.length;j++) {
                        monstres[j].play();
                    }
                } });
```

```
        frame.getContentPane().add(jb, BorderLayout.SOUTH);
        frame.pack();
        frame.setVisible(true);
    }
}

import javax.swing.*;
import java.awt.event.*;
import java.util.*;

class Region extends Observable {
    // relation inverse de la Carte-Region (1-1)
    private Carte carte;
    private int x;
    private int y;

    // relation Carte-Mauhly (0-1) dans ce sens
    private Mauhly mauhly=null;

    public Region(Carte carte, int x, int y) {
        this.carte=carte;
        this.x=x;
        this.y=y;
    }

    public void setMauhly(Mauhly mauhly) {
        this.mauhly=mauhly;
        // signaler le changement d'état
        this.setChanged();
        this.notifyObservers();
    }

    public JComponent display() {
        JComponent c=new IHM();
        // ajouter la vue dans la liste
        // des observeurs
        this.addObserver((Observer)c);
        return c;
    }

    class IHM extends JLabel implements Observer {
        private ImageIcon herbeicon=new ImageIcon("images/herbe.gif");
        private ImageIcon mauhlyicon=new ImageIcon("images/mauhly.gif");

        public IHM() {
            update(Region.this, null);
        }

      // appelé en cas de changement d'état du modèle.
        public void update(java.util.Observable o, Object arg) {
            if (mauhly==null) {
                this.setIcon(herbeicon);
```

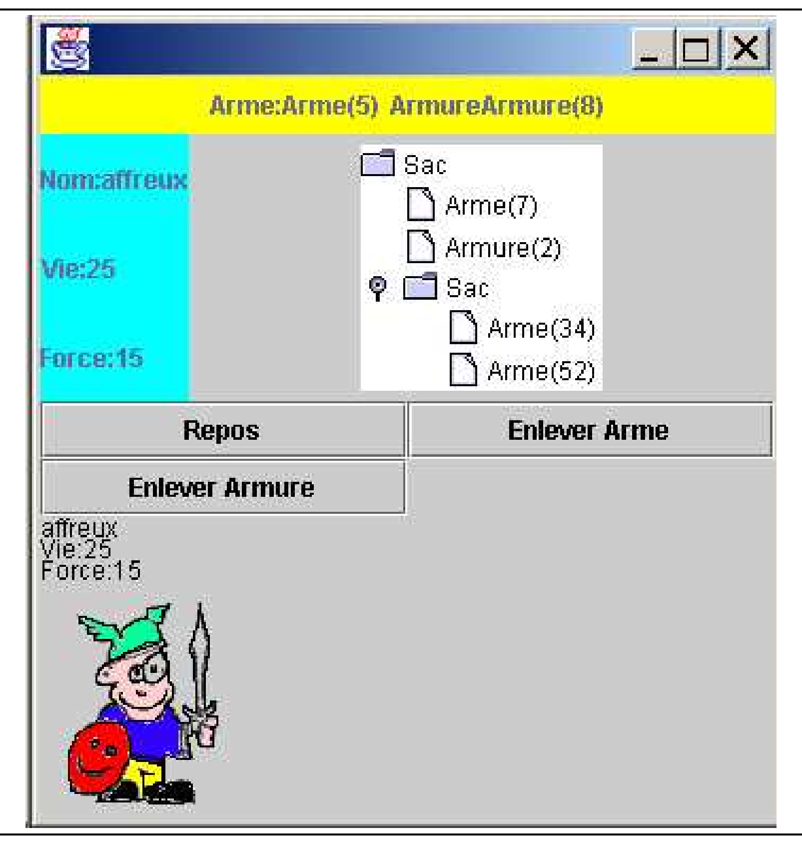
```
                } else {
                    this . setIcon (mauhlyicon);
                }
            }
        }
}
```

9.5.2 Le MVC dans Swing

Nous avons montré deux exemples ou nous avons complétement géré le MVC par nous-même. Le MVC est lui-même très présent spécialement dans librairie graphique Swing. Les composants graphiques de la librairie graphique sont conçus comme des vues de modèles génériques auquel le programmeur doit se conformer pour pouvoir les utiliser.

Nous prenons l'exemple du composant graphique arbre qui est très représentatif. Nous avons revisité l'exemple de l'écran de visualisation du héros en considérant que le sac peut contenir un autre sac. C'est l'exemple type du modèle par composition. De ce fait; il possibre de constuire des hiérarchie de sac qui peuvent être facilement visualisée dans un arbre (cf figure 9.32).

Figure 9.32 Visualiser des arbres



La librarie Swing propose le composant graphique javax.swing.Jtree pour représenter graphiquement des arbres. En fait, la classe “Jtree” est construite comme la vue d'un modèle générique. Ce modèle est défini par une interface: “TreeModel”. Pour pouvoir utiliser un

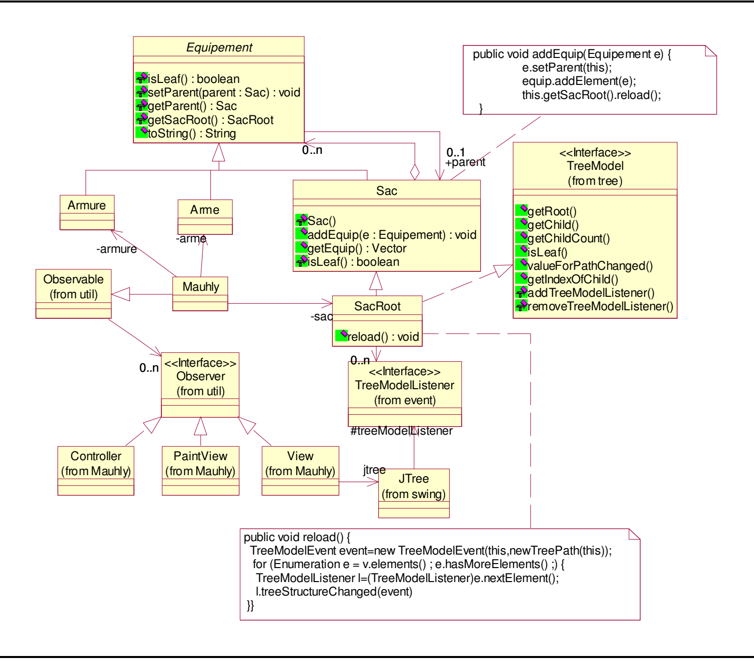
JTree, il faut donc au préalable disposer d'une classe modèle i.e. une classe qui implémante l'interface "TreeModel". La suite est conforme au MVC, on enregistre le "JTree" auprès du modèle et tous changement d'état du modèle sera notifié à la vue qui se remettre à jour automatiquement. Ici, cette remise à jour s'appuie sur l'interface "TreeModel".

Il y a plusieurs façon de disposer d'un modèle d'arbre:

1. Implémenter l'interface "TreeModel" directement
2. Utiliser l'implémentation par défaut fournie dans la librairie Swing

Nous avons décidé d'implanter directement l'interface TreeModel parce que cette solution ne modifie que très peu le modèle. La diagramme de classe de la figure 9.33 montre le schéma général de l'application. Les classe Arme, Armure, Sac et Equipement implante un modèle par composition. La classe Sac représente les noeuds, les classes Armes et Armures les feuilles. La classe Equipement joue le role de la classe abstraite.

Figure 9.33 Diagramme de classes de l'application "sactree"



Le programme suivant montre le code de la classe "SacTree". Cette classe représente l'implantation de l'interface "TreeModel". Cette classe permet de rendre notre composite conforme au modèle attendu par le composant JTree.

```
import javax.swing.tree.*;
import java.util.*;
import javax.swing.event.*;

class SacRoot extends Sac implements TreeModel {
    private Vector v=new Vector();

    SacRoot() {
    }

    public void addTreeModelListener(TreeModelListener l) {
```

```
        v.addElement(l);
    }
    public Object getChild(Object parent, int index) {
        if (((Equipement)parent).isLeaf()) {
            return null;
        } else {
            return ((Sac)parent).getEquip().get(index);
        }
    }
    public int getChildCount(Object parent) {
        if (((Equipement)parent).isLeaf()) {
            return 0;
        } else {
            return ((Sac)parent).getEquip().size();
        }
    }
    public int getIndexOfChild(Object parent, Object child) {
        if (((Equipement)parent).isLeaf()) {
            return 0;
        } else {
            return ((Sac)parent).getEquip().indexOf(child);
        }
    }

    public Object getRoot() {
        return this;
    }
    public boolean isLeaf(Object node) {
        return ((Equipement)node).isLeaf();
    }
    public void removeTreeModelListener(TreeModelListener l) {
        v.removeElement(l);
    }

    public void valueForPathChanged(TreePath path, Object newValue) {
    }

    public void reload() {
        System.out.println("yo");
        TreeModelEvent event=new TreeModelEvent(this,new TreePath(this));
        for (Enumeration e = v.elements() ; e.hasMoreElements() ;) {
            TreeModelListener l=(TreeModelListener)e.nextElement();
            l.treeStructureChanged(event);
        }
    }
}
```

La classe SacTree fournit:

- la gestion des “TreeModeListener” qui sont en fait les vues du modèle dont le composant JTree fera partie.
- Une interface pour permettre au composant JTree de parcourir le composite. Ce sont les méthodes“getChild*”. L’implantation de ces méthodes nous impose de modifier un

minimum le modèle composite. Tous les éléments du composite doivent implémenter "isLeaf" et répondre "oui" si il s'agit d'une feuille et non si il s'agit d'un noeud. La méthode "isLeaf" de la classe "équipement" implante cette méthode en répondant systèmatiquement oui. Cette méthode est redéfinie dans Sac pour répondre "non".

– Un point d'entrée pour commencer le parcours d'arbre : getRoot().

Nous avons implanté grossièrement l'interface TreeModel. Lorsque le composite change, le composite appelle la méthode "reload()" qui redessine entrièment l'arbre graphique. Il est possible de gérer cela plus finement en ne redessinant que le sous-arbre ayant changé.

9.6 Graphisme 2D: les bases


Figure 9.34 Minitortues: version 2D

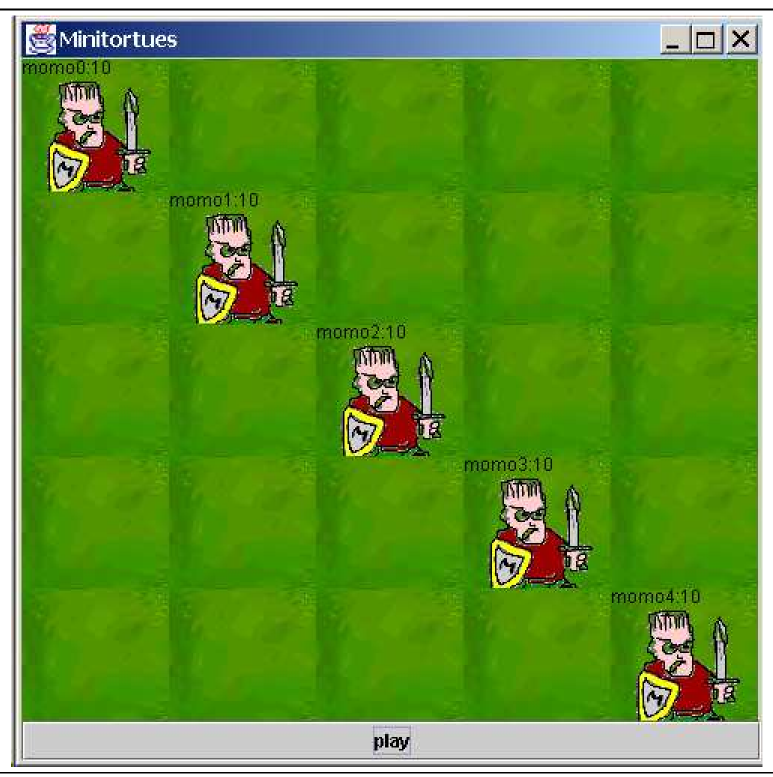
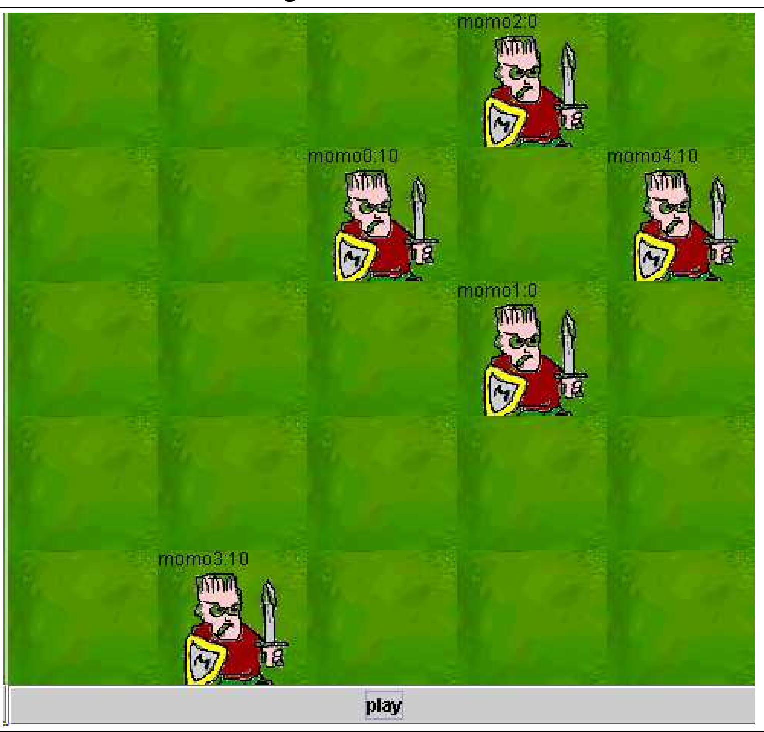


Figure 9.35 Minitortues: elles bougent...



```
import javax.swing.*;
import java.awt.event.*;
import java.util.*;
import java.awt.*;

class Region extends Observable {
    // relation inverse de la Carte-Region (1-1)
    private Carte carte;
    private int x;
    private int y;

    // relation Carte-Mauhly (0-1) dans ce sens
    private Mauhly mauhly=null;

    public Region(Carte carte, int x, int y) {
        this.carte=carte;
        this.x=x;
        this.y=y;
    }

    public Carte getCarte() {
        return carte;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void setMauhly(Mauhly mauhly) {
        this.mauhly=mauhly;
        this.setChanged();
        this.notifyObservers();
    }

    public Mauhly getMauhly() {
        return mauhly;
    }

    public Component display() {
        Component c=new IHM();
        this.addObserver((Observer)c);
        return c;
    }

    class IHM extends Canvas implements Observer {
        private final Image herbeimg=Toolkit.getDefaultToolkit().getImage("images/herbe.gif");

        public IHM() {
            update(Region.this, null);
        }

        public void update(java.util.Observable o, Object arg) {

            this.repaint();
        }
```

Annexe A
Lexique Java

Annexe B

Élements de syntaxe Java

Annexe C
UML